California State University, San Bernardino
CSUSB ScholarWorks

Theses Digitization Project | John M. Pfau Library

2007

Multi-user game development

Cheng-Yu Hung

Follow this and additional works at: https://scholarworks.lib.csusb.edu/etd-project

Part of the Software Engineering Commons

Recommended Citation
Hung, Cheng-Yu, "Multi-user game development" (2007). *Theses Digitization Project*. 3122.
https://scholarworks.lib.csusb.edu/etd-project/3122

This Project is brought to you for free and open access by the John M. Pfau Library at CSUSB ScholarWorks. It has been accepted for inclusion in Theses Digitization Project by an authorized administrator of CSUSB ScholarWorks. For more information, please contact scholarworks@csusb.edu.

MULTI-USER GAME DEVELOPMENT

A Project

Presented to the

Faculty of

California State University,

San Bernardino

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

in

Computer Science

---

by

Cheng-Yu Hung

June 2007

MULTI-USER GAME DEVELOPMENT

A Project

Presented to the

Faculty of

California State University,

San Bernardino

---

by

Cheng-Yu Hung

June 2007

Approved by:

Dr. David Turner, Chair, Computer Science

5/29/07

Date

Dr. Kerstin Voigt

Dr. Ernesto Gomez

ABSTRACT

In the current game market, the 3D multi-user game is the most popular game. To develop a successful 3D multi-user game, we need 2D artists, 3D artists and programmers to work together and use tools to author the game and a game engine to perform the game. Most of this project is about the 3D model development using tools such as Blender, and integration of the 3D models with a level editor and game engine.

This project included the development of a multi-user game that takes place in a 3D world of the computer science department. Basically, the game allows prospective students to meet existing students and faculty in a virtual open house that takes place within the third floor of Jack Brown Hall. Users can walk around Jack Brown Hall and type text messages to chat with each other.

ACKNOWLEDGMENTS

First, I would like to acknowledge my advisor, Dr. David Turner for all the efforts that he had devoted to make this project possible. I would also like to thank Dr. Voigt and Dr. Gomez for serving on my committee and for giving me guidance along the way.

I also thank my classmate Fadi Shihadeh, who gave me a lot of help on 3D modeling, and James Finley, who taught me everything about Blender. I would also like to thank William Herrera, who developed the World Studio level editor and game engine. Also, I thank all the other students working on game development projects for the advice they gave to me. Finally, I would like to thank my family for supporting me and giving me courage to study abroad.

TABLE OF CONTENTS

ABSTRACT iii

ACKNOWLEDGMENTS iv

LIST OF TABLES viii

LIST OF FIGURES ix

CHAPTER ONE: INTRODUCTION

1.1 Purpose 1

1.2 Scope 2

1.3 History of Game Engines 3

1.4 Definition and Abbreviations 6

CHAPTER TWO: SOFTWARE TOOLS AND GAME PRODUCTION

2.1 Overview 8

2.2 Game Production Tools 8

2.2.1 Blender 8

2.2.2 Graphic Editor 9

2.2.3 3D Model Conversion Tool 10

2.2.4 Procedural Terrain Texture Generator 11

2.2.5 Level Editor 12

2.3 Game Production Process 14

2.3.1 Create Art Assets 14

2.3.2 Preprocess Art Assets 17

2.3.3 Produce Level File 21

CHAPTER THREE: USER PROGRAMS

3.1 Overview 22
3.2 Game Client and Game Server Communication 25
3.3 Architecture of the Project Game Engine 27
3.3.1 Game Loop 27
3.3.2 Camera 28
3.3.3 Collision Detection 33
3.3.4 3D Rendering 38
CHAPTER FOUR: 3D MODELING
4.1 Overview 39
4.2 Blender 41
4.3 Construction of the Project World Model 42
4.3.1 Jack Brown Hall 42
4.3.2 Ceiling and Floor 47
4.3.3 Doors 48
4.3.4 Sky 51
4.3.5 Ground 53
4.3.6 Textures 55
4.4 Construction of the Character Model 59
4.4.1 Character Model 59
4.4.2 Armature 63
CHAPTER FIVE: CONCLUSION AND FUTURE DIRECTIONS
5.1 Conclusion 69
5.2 Future Directions 69

REFERENCES 71

LIST OF TABLES

Table 1. Input and Output 3D Model Files 18

Table 2. Texture Information Files 19

Table 3. Camera Movement Hotkeys 28

LIST OF FIGURES

Figure 1. Using Graphic Editor to Create Textures 10

Figure 2. Set Collidable Objects 13

Figure 3. Game Production Process and Tools 14

Figure 4. Terrain Height Map 15

Figure 5. Player Models 16

Figure 6. Game Engine/Server Communication 26

Figure 7. Camera View in the Game Mode 29

Figure 8. Camera Coordinate 33

Figure 9. Plane Equation 35

Figure 10. Detect Collision with Triangles 36

Figure 11. Wireframe of a Cube and a Cylinder 39

Figure 12. Unwrap 3D Objects with Blender 40

Figure 13. Blender's User Interface 41

Figure 14. Top-view with a Background Image 43

Figure 15. Select Vertices 44

Figure 16. Duplicate Vertices 45

Figure 17. Extrude Object 46

Figure 18. Main Jack Brown Hall Model 46

Figure 19. The 3D Model of the Floor 47

Figure 20. Closed Door 48

Figure 21. Opened Door 49

Figure 22. Door Animation Setting 50

Figure 23. Icosphere 51

Figure 24. Flip Normals 52

Figure 25. Subdivide a Plane 53

Figure 26. Proportional Edit Falloff 54

Figure 27. Ground Model 55

Figure 28. Face Select Mode 56

Figure 29. Image Editor 57

Figure 30. Sky Model 58

Figure 31. Subdivided Plane 59

Figure 32. Face Model 60

Figure 33. Complete Face Model 61

Figure 34. Character Model 62

Figure 35. Character Model Textures 63

Figure 36. Assigning Names to the Bones 65

Figure 37. The Complete Armature 66

Figure 38. Assigning Vertices to a Bone 67

Figure 39. Walking Pose 68

CHAPTER ONE

INTRODUCTION

1.1 Purpose

The project is proposed with three purposes in mind. First, the purpose is to get experience in a 3D game development project. In this part, I needed to do research on how a game is developed and what game engines are generally used now. Ultimately, I decided with my advisor to collaborate with other students working on game development under the guidance of Dr. Turner, especially William Herrera, who was the principal architect of the level editor and game engine that I used to develop the virtual open house game. The level editor and game engine was developed using C++, and Blender was used to develop the 3D model of Jack Brown Hall, the players and other 3D objects placed into the virtual world using the level editor. Collaborating with other game developers enabled me to gain experience working with other people on a development project.

Because Blender is a well established, free, open source 3D modeling program, I chose to use this to develop the world model (third floor of Jack Brown Hall). I also experimented with Blender's internal game engine to see whether this could be used to implement the entire game. I discovered that

Blender's game engine is limited in what it can do, so that ultimately, I decided to collaborate with other game developers, who were trying to build a first-person shooter game engine using the C++ language.

With Blender, I was able to build a 3D virtual world of Jack Brown Hall, which is loaded by the game engine, so that users can walk around the area that comprises the computer science department. I coordinated my development of the 3D model with William Herrera, who developed the level editor and game engine code to allow users see avatars of each other in Jack Brown Hall, and send text messages to each other.

1.2 Scope

The scope of this project includes understanding what a game development environment is, and developing an interactive multi-user game through collaboration with other developers. The 3D models are exported to the 3DS file format, which is the most popular 3D model format now. The models can also be used for developing other games, such as first-person-shooter (FPS) games. The models are comprised of several objects, and it is possible to add new objects into the models for further uses.

1.3 History of Game Engines

A long time ago, game developers only cared about how fast they could develop games for the market. Even if these games were still very easy and low quality, it took 8 to 10 months to develop a game on average. One of the reasons was because of technology. Another reason was because they needed to do a lot of low-level coding. With time and experience, developers found out that they could reuse some code of previous games and make them into frameworks, so that they could speed up the development time and reduce costs. This was the first concept of game engines.

Every game has its own game engine, but not every game engine can be a standard game engine. The most popular game engines are 3D games, especially first person shooter games. Even the most famous 2D game engine, the Infinity Engine, which produced Baldur's Gate (1998) and Icewind Dale (2000), is limited to develop Advanced Dungeons & Dragons (ADnD) games. This is also a major reason that the game engines of sports games, simulation games, and strategy games are rarely licensed in the game market.

Game engines started in the 1990s. In 1992, the first 3D first-person shooter game, Wolfenstein 3D, was published by Apogee Software. The game engine of Wolfenstein 3D was

developed by John Carmack, who is generally acknowledged as the most important game programmer in 3D game engines. One year later, he developed the Doom engine, which is the first game engine for licensing. Doom was published by id Software, the first company succeeded in making money on licensing their game engine to other company. Later on, many famous games used the Doom engine, such as Heretic (1994) and Hexen (1995). At that time, id Software became the lead of the 3D game engine.

At the same time, another game programmer of 3D Realm, Ken Silverman, developed a game engine, the Build engine. Like the Doom engine, the Build engine is a game engine for first-person shooter game. However, it included more motions than the Doom engine. Jump, 360-degree view, squat, and swim were included in the Build engine. By licensing the Build engine, 3D Realms got over one million dollars benefit.

One year later, the Quake engine was created for Quake in 1996. The Quake engine is written by id Software and John Carmack did most of the code. Soon, he developed an advanced version engine, the Quake II engine, for Quake II. The Quake II engine supported for hardware-accelerated graphics and OpenGL technique. With these techniques, the Quake II engine enhanced the quality of images. For this reason, more and more games used the Quake II engine, such as Heretic II (1998), SiN

(1998), Kingpin: Life of Crime (1999), Soldier of Fortune (2000), and Anachronox (2001). With the success of the Quake II engine, id Software seemed to become the largest company of developing game engines.

However, it is good to have some competition. Another company, Epic Games, developed the Unreal Engine for their first-person shooter game, Unreal, in 1998. The Unreal engine could be the most wide-used game engine so far. Not only used by many famous games, the Unreal engine also used for education, architecture, and other fields.

At that time, the graphic performance was hardly to get any progress. Programmers knew that they needed to develop other functions to enhance their games engines. For this reason, the Half-Life 1 Engine was developed. The Half-Life 1 Engine was the heavily modified Quake Engine for their science fiction FPS game, Half-Life (1998). The big evolution of this engine was its scripted sequence technique. The scripted sequence could make the player experience and watch a cut scene when the player triggered it. With the scripted sequence, a first-person shooter game would not be a hollow shooting contest any more.

Another big evolution of game engines in 1998 was Artificial Intelligence. Looking Glass Studios using the Dark

Engine developed a single player stealth-based computer game, Thief: The Dark Project (1998). The Dark Engine had a great achievement on Artificial Intelligence. For example, the NPCs in the game could detect the player's location by the footsteps, and they would have poor eyesight in the dark area. The player had to hide in the dark and keep quiet to prevent that NPCs noticed. These ideas inspired other game programmers to focus on Artificial Intelligence.

After 2000, developing game engines had two different subjects. First one was adding more scripted sequences and enhancing Artificial Intelligence. The other one still focused on the multi-player game mode. No mater what subject it is, the objective is the same: to make the game more interesting.

1.4 Definition and Abbreviations

Texture - A texture is a bitmap image used to apply a design onto the surface of a 3D computer model.

BSP - A Binary Space Partition (BSP) is a technique for determining polygon order and therefore visibility by cutting a world space into convex regions. Because each cut splits the world into two sub-regions, they use the word "binary".

CSG - Constructive Solid Geometry (CSG) is a technique for defining a detailed space by building it up gradually with simple shapes.

UV Mapping - This is the most flexible way of mapping a 2D texture over a 3D object. It takes a three-dimensional mesh in (x, y, z) coordinates, and maps it to a flat two-dimensional image in (u, v) coordinates.

MMOG - Massively-multiplayer online games

ADnD - ADnD (Advanced Dungeons & Dragons) is a kind of role-playing game.

Level Editor - A level editor is a program that is used to create game worlds for a specific game engine.

OpenGL - OpenGL (Open Graphics Library) is an application programming interface for programming graphics components of software programs.

NPC - A NPC (Non-player character) is a character in a computer game that is controlled by the computer.

CHAPTER TWO

SOFTWARE TOOLS AND GAME PRODUCTION

2.1 Overview

Developing a game is a team work. Basically, there are at least three roles in the developing team: programmers, game designers, and 3D modelers. Developers need to use different tools for different purposes. In this project, some tools were used by 3D modelers, and some tools were developed by programmers to produce games.

2.2 Game Production Tools

There are several different types of tools used in this project. Some of the tools are available for free; other tools were built specifically for this project. The already available tools include Blender and GIMP. The tools that were built specifically for this project include the WorldStudio level editor, the 3D model conversion tool, and the terrain generator.

2.2.1 Blender

Blender is the 3D modeling software used in this project. It is free open source software. It is able to run on several different operation systems, including Microsoft Windows,

GNU/Linux, Mac OS X, and FreeBSD [6]. There are more detailed descriptions in Chapter Four.

2.2.2 Graphic Editor

The GNU Image Manipulation Program (GIMP) is a free graphic editor, which can run on GNU/Linux systems, Windows systems, and Mac systems. GIMP has many capabilities. It can be used for digital painting, photo retouching, and image format converting [9]. In this project, GIMP is mostly used for digital painting and creating textures. The picture below is the screenshot of using GIMP to create a texture for a character model.

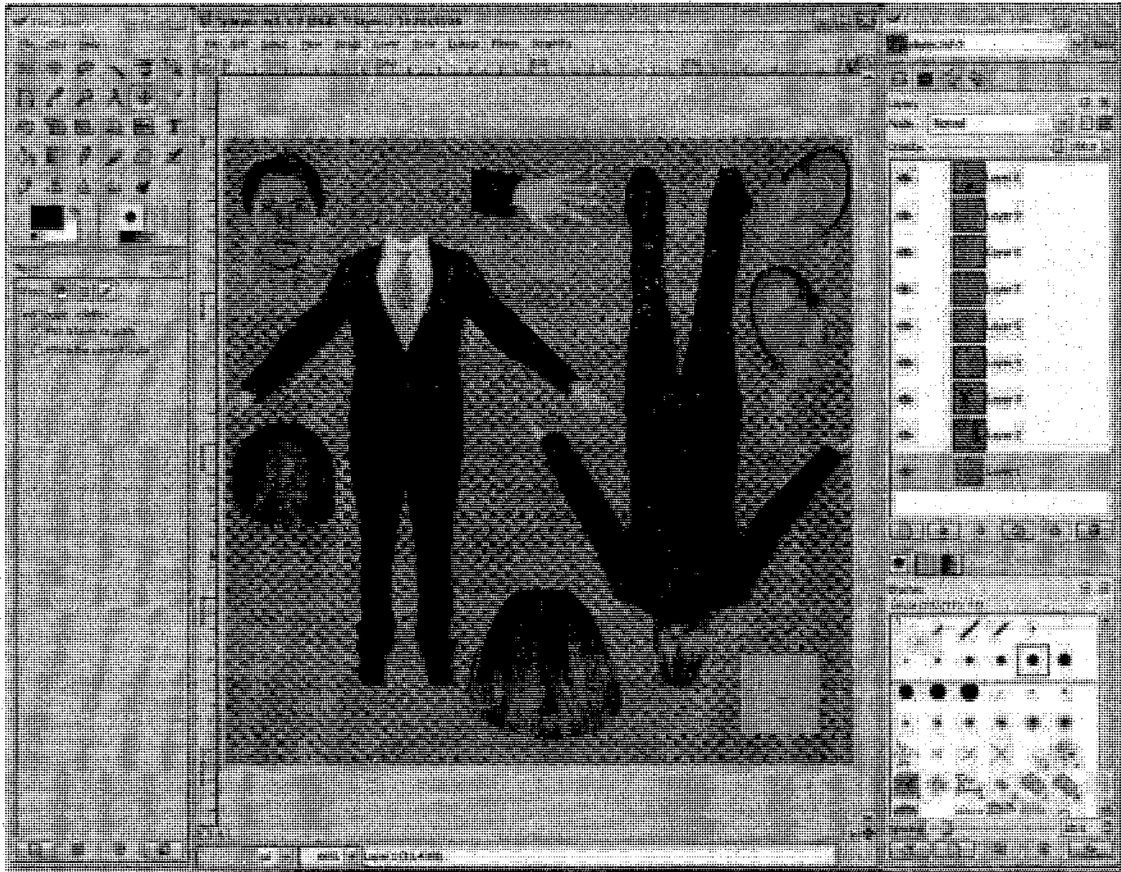

Figure 1. Using Graphic Editor to Create Textures

2.2.3 3D Model Conversion Tool

The 3D model conversion tool is a program that converts 3D models that are built with 3D modeling tools (such as Blender) into a file format that is loaded by the level editor. Currently, the conversion tool is used to convert 3D Studio Max (3DS) files into the format loaded by the game.

To run the 3DS conversion tool, start the executable 3ds_2_worldstudio.exe. The program displays a file selection

window. In the file selection window, the user selects the 3DS file to convert. The program reads the selected file, and saves the converted data into a file with name equal to the file name of the input file (minus its extension) appended with .3dsactor.dat. For example, if the selected input file is named sky.3ds, the generated output file will be named sky.3dsactor.dat.

2.2.4 Procedural Terrain Texture Generator

The procedural terrain texture generator is a command line program that reads three input files and outputs a procedural terrain texture that can be loaded by the WorldStudio level editor. The outputted file is a raw 2D image (24 bit color pixels) that is used to define the position and color of the terrain.

The generator takes three files as input, and outputs a single file. The three input files include the following:

- Definition of a rock texture (an array of raw RGB pixel data, 256 by 256)
- Definition of a dirt texture (an array of raw RGB pixel data, 256 by 256)
- Array of height values, called a height texture (an array of single-byte grey scale values, 256 by 256)

The array dimensions are 256 by 256. The color values represent the full extent of the game world. Because the game may have different dimensions, the array of colors is expanded or contracted as needed to cover the world exactly.

The height texture is created in a 2D image editor (such as Photoshop). Black represents ground level, and white represents maximum height. Therefore, when a level surface is desired, the game designer colors the area as black, and when a mountainous area is desired, the game designer colors the area in varying shades of gray.

The terrain generator combines the three inputs, and outputs an array of raw RGB pixel data. Calculation of an element in the output array is as follows:

$$t_{ij} = d_{ij}(1 - h_{ij}) + r_{ij}h_{ij}$$

Where t_{ij} is the resulting texture color value for the terrain, d_{ij} is the dirt color value, h_{ij} is the height value normalized to fall within [0,1], and r_{ij} is the rock color value.

2.2.5 Level Editor

The level editor used in this project is called WorldStudio. It reads the 3D models and then allows the game designer to perform the following actions:

- Select the collidable objects
- Create collision boundaries

■ Select the invisible objects (see-through windows)

The game designer saves the world definition in a file that the game client loads.

The picture below shows the screenshot of setting Jack Brown Hall model as a collidable object.

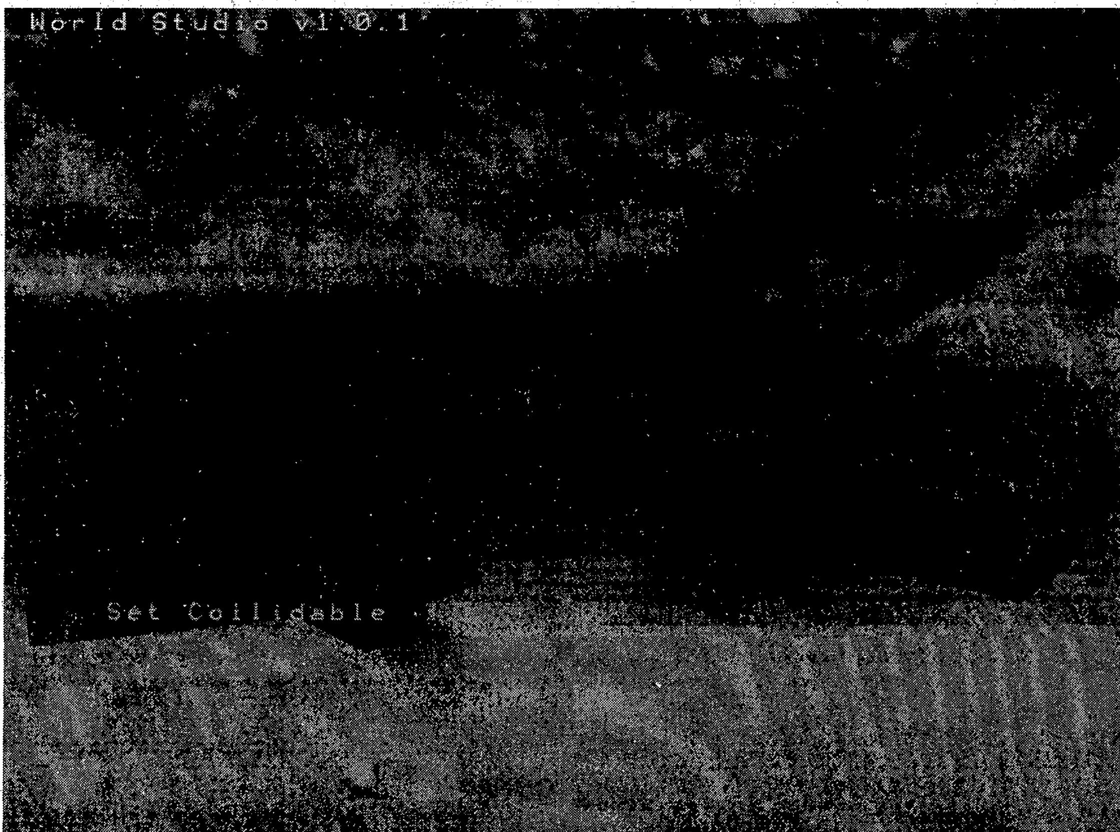

Figure 2. Set Collidable Objects

2.3 Game Production Process

The game production process in this project has three steps. This section describes the process used by the game development team to create a game. The following picture shows the diagram of game production process and tools.

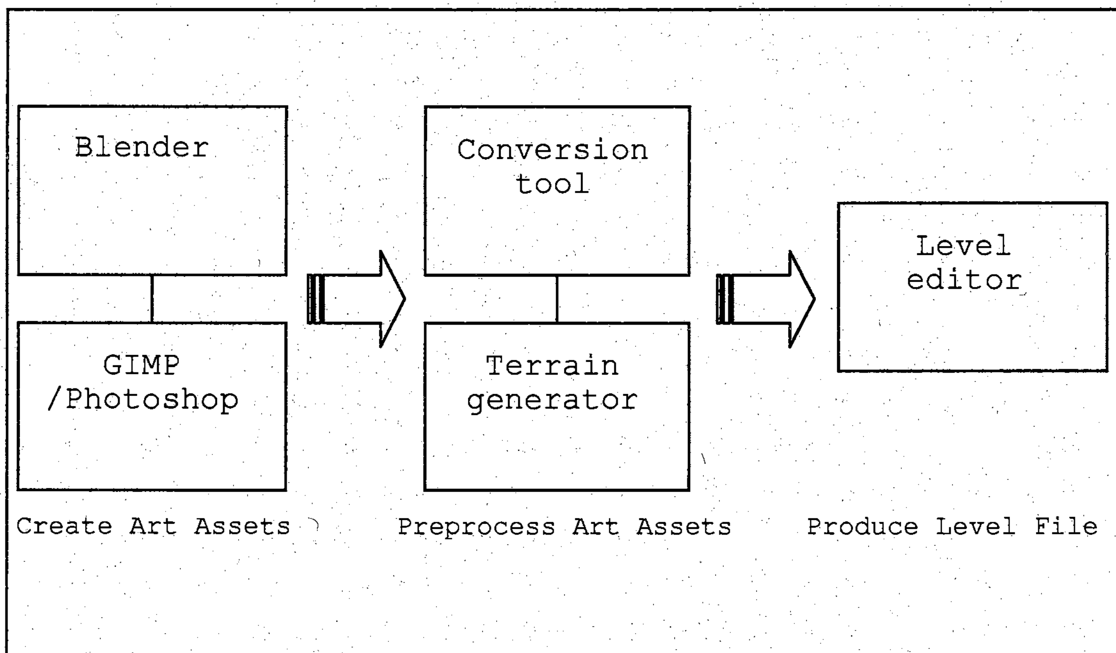

Figure 3. Game Production Process and Tools

2.3.1 Create Art Assets

In the first step, an artist creates the following:

- A 3D model of a building
- A terrain height map

- A collection of 3D models that represent the animation frames to render a player
- Textures to apply to 3D models
- Rock texture and dirt texture

A 3D model of a building is created in Blender and exported in 3D Studio Max (3DS) file format. The current 3D building model file is jb_full.3DS.

A terrain height map is a grey scale image file created in Photoshop and saved as RAW file format. The following picture is the current map file, map.raw. The black pixels represent a height of 0, which is level ground. The more white a pixel has, the higher that point is vertically.

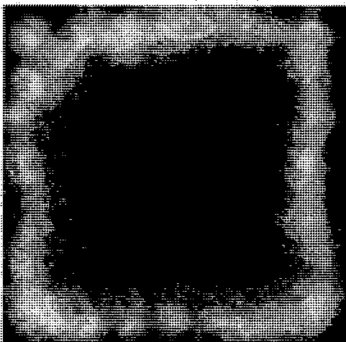

Figure 4. Terrain Height Map

A collection of 3D models that represent the animation frames to render a player are also created in Blender. The

details of creating player models will be described in Chapter Four. There are four different poses of player models which are used to present a walking animation. These files are avatar1.3DS, avatar2.3DS, avatar3.3DS, and avatar4.3DS. Figure 5 shows the complete models of each pose.

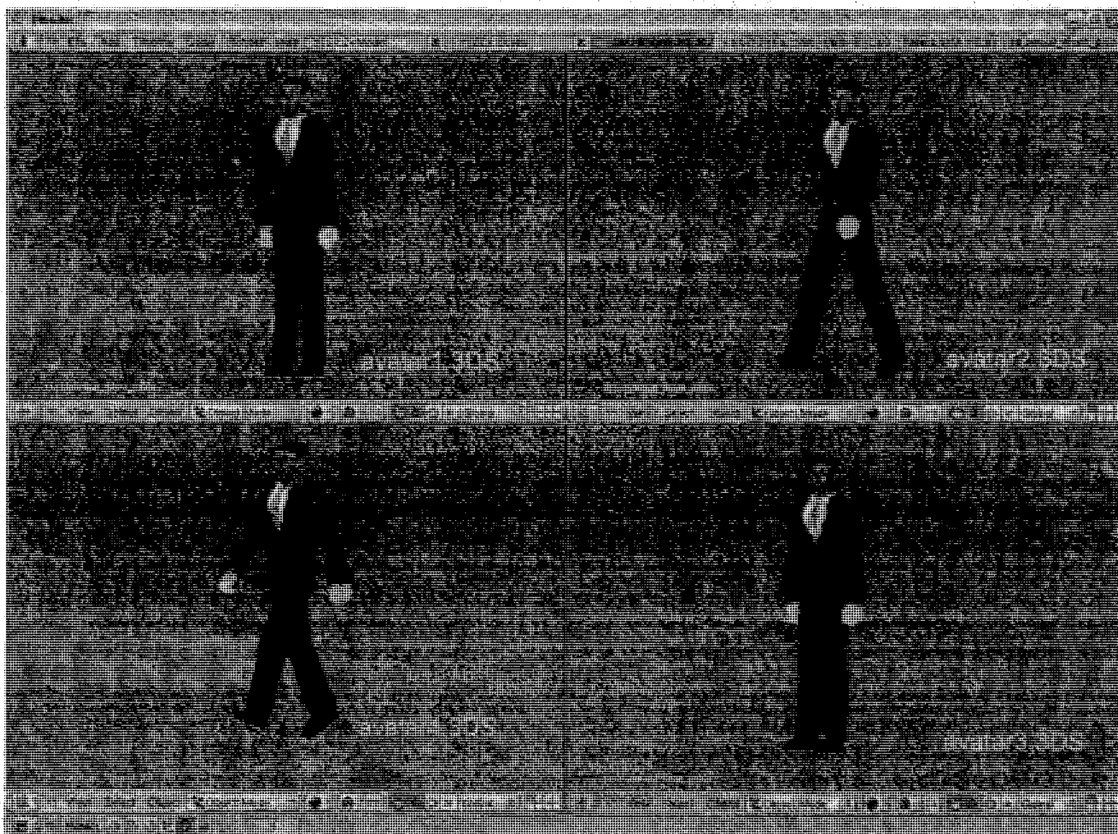

Figure 5. Player Models

Textures are image files, which are created in GIMP or Photoshop. Currently, we are store textures in the following formats:

- TGA
- RAW
- JPEG
- BMP

2.3.2 Preprocess Art Assets

In the second step, the game designer uses conversion tools to convert textures and 3D models into a file format used by the level editor and game engine. The game designer preprocesses the art assets as follows:

- Apply the procedural texture terrain generator to the rock texture, dirt texture and height map to produce a procedural texture for the terrain.
- Apply the 3DS-to-worldstudio command to the player models and the building model to produce the actor files that represent objects in the virtual world.

We use the convention tools to convert 3DS files into the file format used by the level editor and game engine. The following table shows the list of exported files in this project.

Table 1. Input and Output 3D Model Files

Model Type	Input File	Output File Name
3D building model	Jb_full.3DS	jb_full.3dsactor.dat
Sky model	sky.3DS	sky.3dsactor.dat
Player models	avatar1.3DS avatar2.3DS avatar3.3DS avatar4.3DS	avatar1.3dsactor.dat avatar2.3dsactor.dat avatar3.3dsactor.dat avatar4.3dsactor.dat

The 3D model conversion tool also exports a text file that contains the name and location of the images that are used to texture the 3D objects. We need to manually edit the text file so that it can be read by the level editor and associated with the correct UV coordinates. For example, after running the 3DS conversion tool with selected file sky.3ds, there will be an output text file named materials.txt. The file content is "jpg_texture None_sky.jpg", where "jpg_texture" means that the texture is in JPEG file format, "None_" means that the 3D model is not using any material name, and "sky.jpg" is the texture file name. Since we don't use any material name in this 3D model, we have to delete "None_". Also, because we put the texture file in the DATA/SKY folder, we need to change "sky.jpg" to "data/sky.jpg". Finally, the file contents will be "jpg_texture data/sky/sky.jpg". The following table is the texture information in this project. Note that the order of

texture file names is important. If a texture in this list is in position n, then the nth surface in the corresponding wire-frame model should be colored with this texture.

Table 2. Texture Information Files

File Name	File Content
materials.txt (3D Building Model)	jpg_texture data/jackbrownhall/gate.jpg jpg_texture data/jackbrownhall/door2.jpg jpg_texture data/jackbrownhall/wall1.jpg jpg_texture data/jackbrownhall/wall1.jpg jpg_texture data/jackbrownhall/ceiling.jpg jpg_texture data/jackbrownhall/math.jpg jpg_texture data/jackbrownhall/r360.jpg jpg_texture data/jackbrownhall/r359.jpg jpg_texture data/jackbrownhall/r358.jpg jpg_texture data/jackbrownhall/r356.jpg jpg_texture data/jackbrownhall/door3.jpg jpg_texture data/jackbrownhall/r349.jpg jpg_texture data/jackbrownhall/r347.jpg jpg_texture data/jackbrownhall/r345.jpg jpg_texture data/jackbrownhall/r343.jpg jpg_texture data/jackbrownhall/r341.jpg jpg_texture data/jackbrownhall/r339.jpg jpg_texture data/jackbrownhall/r337.jpg jpg_texture data/jackbrownhall/r335.jpg jpg_texture data/jackbrownhall/r333.jpg jpg_texture data/jackbrownhall/r331.jpg jpg_texture data/jackbrownhall/r329.jpg jpg_texture data/jackbrownhall/r327.jpg jpg_texture data/jackbrownhall/r325.jpg jpg_texture data/jackbrownhall/r323.jpg jpg_texture data/jackbrownhall/r321.jpg jpg_texture data/jackbrownhall/r319.jpg jpg_texture data/jackbrownhall/r317.jpg jpg_texture data/jackbrownhall/r315.jpg jpg_texture data/jackbrownhall/r313.jpg jpg_texture data/jackbrownhall/r311.jpg jpg_texture data/jackbrownhall/r310.jpg jpg_texture data/jackbrownhall/r312.jpg

	jpg_texture data/jackbrownhall/r314.jpg jpg_texture data/jackbrownhall/r316.jpg jpg_texture data/jackbrownhall/r318.jpg jpg_texture data/jackbrownhall/r320.jpg jpg_texture data/jackbrownhall/r322.jpg jpg_texture data/jackbrownhall/r324.jpg jpg_texture data/jackbrownhall/r326.jpg jpg_texture data/jackbrownhall/r328.jpg jpg_texture data/jackbrownhall/r330.jpg jpg_texture data/jackbrownhall/r332.jpg jpg_texture data/jackbrownhall/r334.jpg jpg_texture data/jackbrownhall/r336.jpg jpg_texture data/jackbrownhall/r338.jpg jpg_texture data/jackbrownhall/r348.jpg jpg_texture data/jackbrownhall/r346.jpg jpg_texture data/jackbrownhall/r344.jpg jpg_texture data/jackbrownhall/r342.jpg jpg_texture data/jackbrownhall/r340.jpg jpg_texture data/jackbrownhall/black.jpg jpg_texture data/jackbrownhall/floor.jpg jpg_texture data/jackbrownhall/alumi.jpg jpg_texture data/jackbrownhall/eve_r.jpg jpg_texture data/jackbrownhall/eve_m.jpg jpg_texture data/jackbrownhall/eve1.jpg jpg_texture data/jackbrownhall/eve_l.jpg jpg_texture data/jackbrownhall/door1.jpg jpg_texture data/jackbrownhall/board.jpg jpg_texture data/jackbrownhall/CS.jpg jpg_texture data/jackbrownhall/door5.jpg jpg_texture data/jackbrownhall/door4.jpg jpg_texture data/jackbrownhall/eve_2.jpg jpg_texture data/jackbrownhall/eve2.jpg jpg_texture data/jackbrownhall/board2.jpg jpg_texture data/jackbrownhall/alumi.jpg
Sky.txt (Sky model)	jpg_texture data/sky/sky.jpg
avatars.txt (Player model)	jpg_texture data/player.jpg jpg_texture data/shirt.jpg

2.3.3 Produce Level File

The third and final step in game production is to use WorldStudio to produce a level file. The level file contains the definition of the game, including the art, and is loaded by the engine when the user runs the game engine to play a game.

Inside WorldStudio, the designer does the following:

- Load 3D objects
- Define collision boundaries
- Select a brush and add its objects to the game world
- Set objects to visible or invisible

Currently, there are 3 brushes, which are used to create boxes, doors, walls, floors, ceilings and a variety of useful generic shapes.

CHAPTER THREE

USER PROGRAMS

3.1 Overview

An engine in a car determines the performance, the speed and the stability of the car. Similarly, a game engine controls the plot, stages, art rendering, music, and operation of a computer game. Game engines are the leaders of games and bind all the elements to work together. No matter what game genre it is, there is a game engine inside the code of a game.

After continuously developing for many years, a game engine becomes a complex system that includes many components, such as lighting, 3D modeling, animation, a render engine, a physics engine or collision detection, sound, scripting, animation, artificial intelligence, networking, and a scene graph.

Lighting is a very important part in rendering, standing equal to modeling, materials and textures. All the lighting effects are controlled by game engines. A simple model could become very realistic if the 3D artist skillfully uses the lighting effects. However, lighting is a very difficult topic in 3D modeling. In the real world, even if there is only a single light source, such as a lamp or the sun, the light will be

re-irradiated all over the scene and make shadows soft. Therefore, the shadowed regions are not totally black, but partially light.

Animation is also an important part in game engines. In the current game engines, there are two different types of animation. The first one is a static animation in which the game engine renders a sequence of pre-recorded snapshots of the object in different forms. The second one is called ragdoll physics in which the game engine renders the object in progressively different forms in accordance with an approximation to physical laws. The game in this project utilizes static animation procedure. However, a bone skeleton was created for the player model, which was used to generate snapshots of the model in different positions.

A physics engine performs a rule for the movement of objects. For example, when an object jumps, the gravity in this physics engine will determine how height it can jump and how fast when it falls down. Also, a physics engine can determine the trajectory of a bullet.

Collision detection is the major functionality of a physics engine. It can detect all edges of objects in the game. When two objects hit each other, this technique will prevent one object from passing though another object. It means that

when a player in the game collides with a wall, he won't pass though the wall or knock down the wall because collision detection will base the result on the property of the player and the wall to determine their location and action after collision.

Rendering is also one of the most important functionalities in game engines. After building a 3D model, artists put different textures on different faces of objects, like skins on bones. Rendering engines calculate all 3D models, animation, lighting, and other effects and then display the result on the monitor. Rendering engines is the most complex component in game engines. The performance of rendering engine directly determines the quality of the game.

Game engines also have one important function which is communication between users and computers. For example, keyboards, mouse, or joysticks are generally used as input tools and game engines will handle signals from these tools. Game engines of MMOG (massively-multiplayer online games) could also administer the communication between clients and servers.

Artificial Intelligence techniques are also used in some game engines for some purposes, such as path finding, steering, and finite state machines. Alexander Nareyek, director of the

Interactive Intelligence Labs at the National University of Singapore, believes that artificial intelligence is the next big thing of game engines. He also indicates that Artificial Intelligence in games makes games more interesting and interactive. For example, automated storytelling is one idea to make games interesting. Most games have a main storyline and some limited branch stories. If the player can choose a good or evil path and the storyline will also be changed after the player chose the path, it makes the game more interesting and interactive.

3.2 Game Client and Game Server Communication

To play a game, an instance of the game server needs to be run, and an instance of the game client (game engine) needs to run for each player in the game. When the game clients start running, they will try to load the level file that defines the game world, which was developed in the WorldStudio level editor. The game engine is the program that runs on the user's machine, and allows the user to play the game.

Each instance of the Virtual Open House game sends position data for its player. The game engine sends the position data it receives from a given player to all other players. This allows each game instance to render each player

in the virtual world. Also, the game engine is in charge of sending text messages between players.

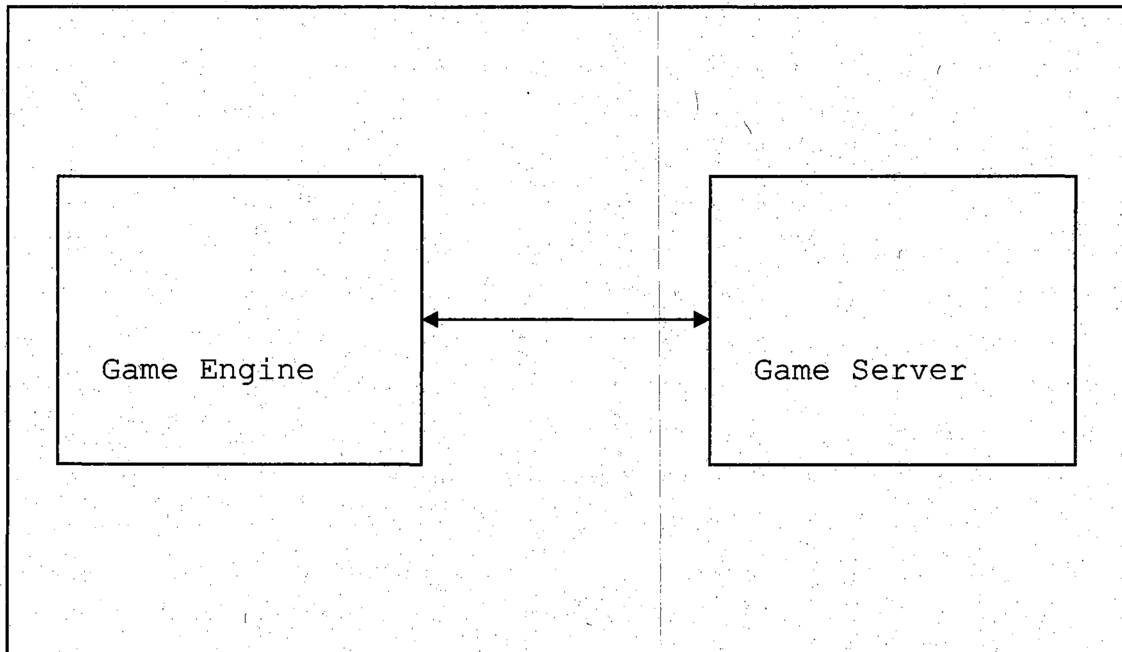

Figure 6. Game Engine/Server Communication

All communication between engine instances is done through the game server. The communication process is as follows:

- A user executes the game engine.
- The engine makes a TCP connection to the server, which it maintains until the user terminates the engine.
- The server assigns an integer identifier to each player.

- The engine sends player position data to the server continuously, which the server broadcasts to all other players.
- The engine sends text messages to the server as the user executes this function.

Position data packets include the following:

- World coordinates of player (x,y,z)
- Rotation of player (rx,ry,rz)
- Speed of player in the direction he or she is facing

Text chat data packets include the following:

- A string representing the text message

3.3 Architecture of the Project Game Engine

3.3.1 Game Loop

The game loop is an infinite loop that performs the following functions:

- Render the visual elements to the screen
- Update the position of opponents when position data is received from the server
- Enforce collision constraints, so that players do not pass through walls)
- Process keyboard input to control the user's character and to send text messages

3.3.2 Camera

The camera is a viewpoint from the player to the 3D world. In this game engine, the camera represents the location of the player in (x, y, z) coordinates, and also angles of view. It means that a player can rotate the camera around the y-axis to simulate the action of turning the head and also change the camera angle between the x-y plane and the line of vision to simulate the action of raising or lowering the head.

We use keyboard and mouse to control the movement of the camera (player). The following table shows the hotkey setting of this game engine.

Table 3. Camera Movement Hotkeys

Mode	Hotkey	Function
3D View Mode	Arrow Key Up (↑)	Move forward
	Arrow Key Down (↓)	Move backward
	Arrow Key Left (←)	Move left
	Arrow Key Right (→)	Move right
	Mouse Left + Right Button	Move forward
	Mouse Cursor	Rotate
Game Mode	W	Move forward
	S	Move backward
	A	Move left
	D	Move right
	Mouse Cursor	Rotate

When a user presses these hotkeys, the game engine will calculate the new position of the player based on the direction and the velocity.

First, we define the screen size as 640x480, so the center of the screen is (320,240). By moving the cursor, the user rotates the camera around the x and y axes. The program maintains these angles with variables rx and ry, where rx is the vertical angle that simulates the action of raising or lowering the head, and ry is the horizontal angle which simulates the action of turning the head and the body.

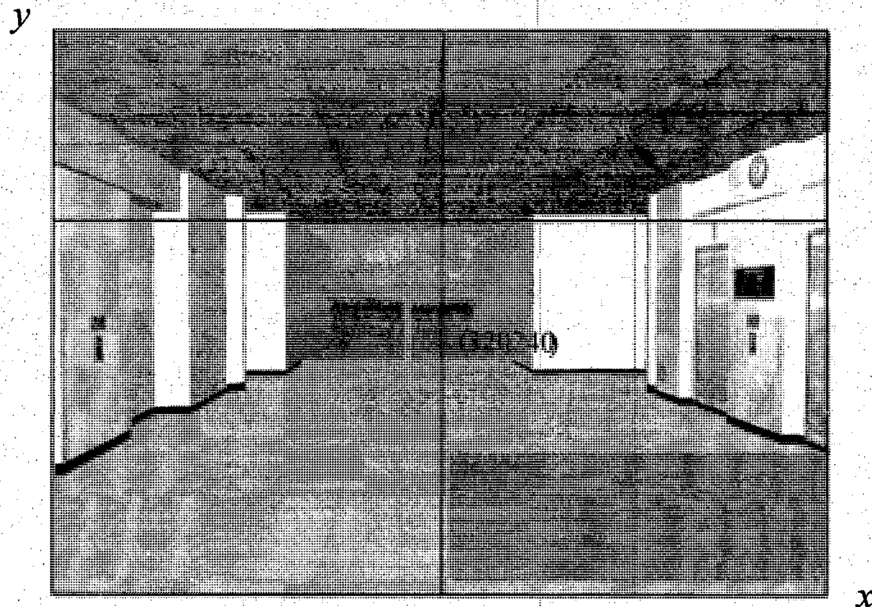

Figure 7. Camera View in the Game Mode

Figure 7 shows the camera view in the game mode. If the cursor is not at (320,240), the camera angle will change. The following pseudo code shows how we rotate the angle by the x-y coordinate of the cursor.

```
if (mouse.x ≠ 320 or mouse.y ≠ 240)
{
  if ((mouse.x-320) > 0)
      ry += 2.0;
  if ((mouse.x-320) < 0)
      ry -= 2.0;
  if ((mouse.y-240) > 0)
      rx += 1.5;
  if ((mouse.y-240) < 0)
      rx -= 1.5;
  if (rx > 45.0)
      rx = 45.0;
  if (rx<-45.0)
      rx=-45.0;
}
```

After getting the angles of rx and ry, the next step is to get the velocity of the player. When the user presses one of the hotkeys in the Table 3, the velocity of the player will

increase 0.05, and the maximum of the velocity is 0.3. Also, different hotkeys have different directions. The following code shows how we define the hotkeys.

```
//keyboard input
switch(windows_msgs->wParam)
{
  case 'a':
  case 'A':
  sy = -90.0;   //move left
     if( velocity < 0.3 )
        velocity+=0.05;
     break;
  case 'd':
  case 'D':
  sy = 90.0;   // move right
     if( velocity < 0.3 )
        velocity+=0.05;
  break;
  case 'w':
  case 'W':
     if( velocity < 0.3 )
        velocity+=0.05;
  break;
```

```
    case 's':
    case 'S':
        if( velocity>0 )
            velocity=0;   // stop
            velocity+=-0.05;     // move backward
    break;
}
```

With the velocity and the angles, rx, ry and sy, the engine can calculate the next position of the player. The equations are:

$$x = x_0 + v \cdot \cos(rx) \cdot \cos(ry + sy)$$

$$z = z_0 + v \cdot \cos(rx) \cdot \sin(ry + sy)$$

Where (x,z) is the new position coordinate, (x_0, z_0) is the original position coordinate, v is the velocity, rx and ry are camera angles, and sy is the direction angle. The following picture shows how these two equations come out.

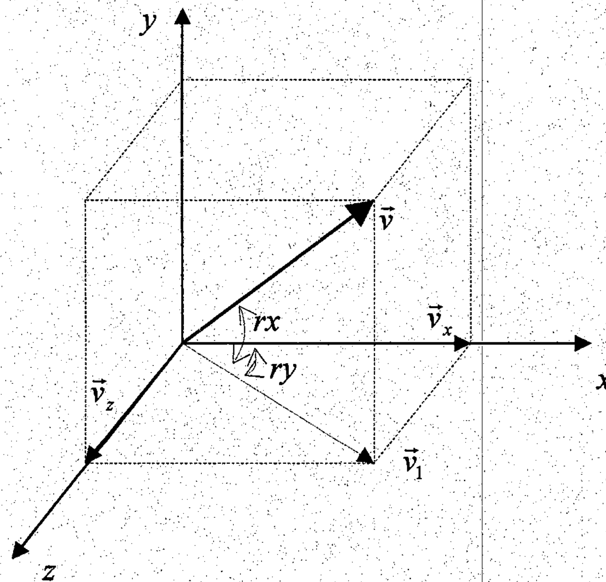

Figure 8. Camera Coordinate

After getting the new position of the player, the engine will run collision detection to check if the player can move to the new position.

3.3.3 Collision Detection

Collision Detection in this project has three steps as follows:

- Use a quad tree to select those objects that are close to the player.
- Check if the path of the player passes through a plane containing any one of the triangles that comprise the object.

- For those triangles selected in the previous step, check if the path of the player passes through the triangle.

The first step is to use the quad tree to check which objects are possible to collide with the player. If the object is too far away to collide, there is no need to do next step.

If the object is inside the radius of the player, we will go to the second step. First, we know that the plane equation is: $ax+by+cz+d=0$, where a, b, c and d are real numbers, $\vec{n}=(a,b,c)$ is the normal vector to the plane, and a, b, c are not all zero. If there is a point $Q=(x,y,z)$ outside the plane, it means $ax+by+cz+d\neq0$. If $ax+by+cz+d>0$, Q is on the front side of the plane. Otherwise, if $ax+by+cz+d<0$, it means Q is on the other side of the plane.

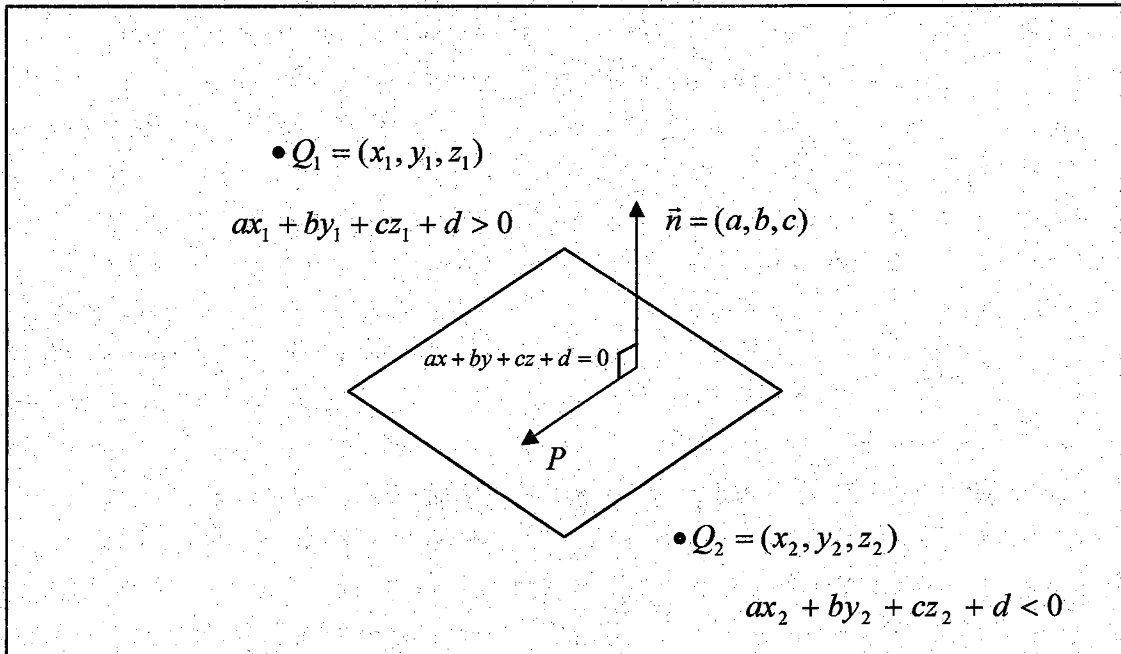

Figure 9. Plane Equation

Each object is comprised of many triangles, and so the algorithm needs to consider many planes. Suppose that Q_1 is the original position of the player, and Q_2 is the new position of the player. If Q_1 and Q_2 are on different sides of the plane, it means that it is possible that the player hits the object when he or she moves from the original position to the new position. The following pseudo code shows the logic to check if the positions are on different sides of the plane:

```
bool check_position(Q1, Q2)
{
check = false;
```

```
before = ax₁+by₁+cz₁+d;
after = ax₂+by₂+cz₂+d;
if ((before<0 and after>0) or (before>0 and after<0))
  check = true;
return check;
}
```

The third step is to check whether the player will hit the candidate triangles selected in the previous step.

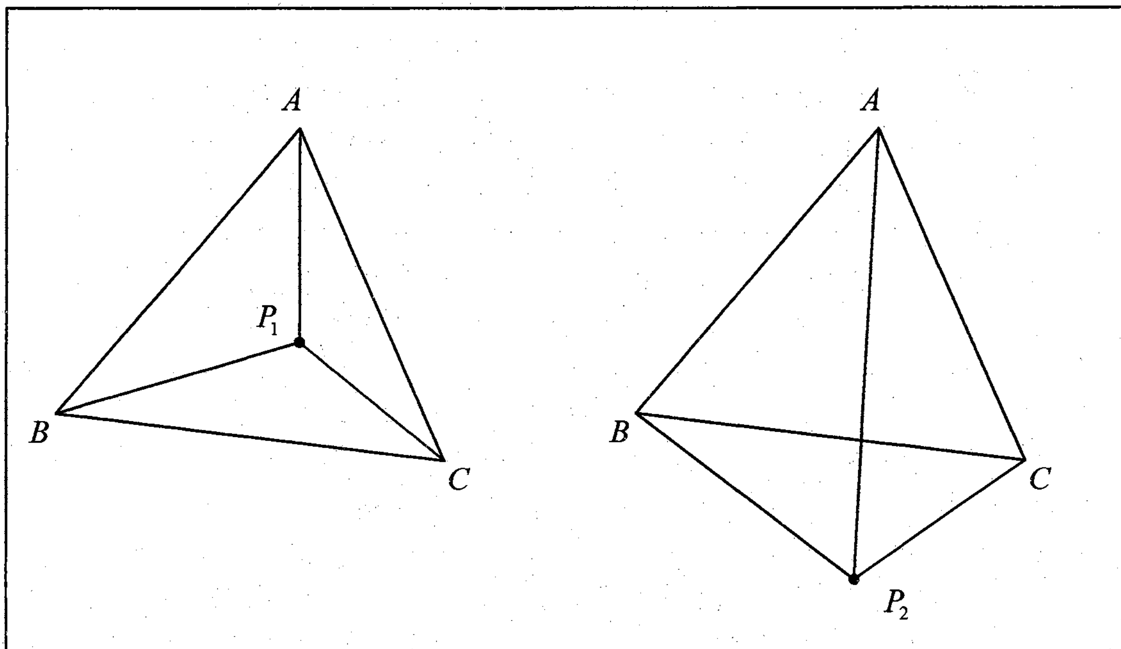

Figure 10. Detect Collision with Triangles

Let p1 and P2 be collision points on the plane containing triangle ABC. Figure 10 shows that if the collision point on

the plane is P_1, and P_1 is inside a triangle, which is formed by three vertices of the object, then $\angle AP_1B + \angle BP_1C + \angle CP_1A = 2\pi$. Otherwise, if the point is outside the triangle, $\angle AP_2B + \angle BP_2C + \angle CP_2A < 2\pi$. It can be proved as following:

$$\begin{aligned}
&\because \angle AP_2B + \angle CP_2A = \angle BP_2C \\
&\therefore \angle AP_2B + \angle BP_2C + \angle CP_2A = 2\angle BP_2C \\
&\because \angle BP_2C + \angle P_2BC + \angle BCP_2 = \pi \\
&\therefore \angle BP_2C < \pi \\
&\therefore \angle AP_2B + \angle BP_2C + \angle CP_2A < 2\pi
\end{aligned}$$

However, we don't actually calculate the collision point; we approximate it by the new position of the player, and we check to see if the sum of angles is almost equal to 2π (6.28318531). It means the new position of the player is very close to the object and will hit inside the triangle. The following pseudo code shows how we detect whether the point is inside the triangle ΔABC.

```
bool mathVertexInTriangle(P, A, B, C)
{
#define epsilon = 0.1;
```

vector0 = $\frac{(P-A)}{\overline{PA}}$; // normalized vector $\overrightarrow{AP}$

vector1 = $\frac{(P-B)}{\overline{PB}}$; // normalized vector $\overrightarrow{BP}$

```
vector2 = (P-C)/|PC|; // normalized vector CP

angle = arccos(vector0 • vector1) + arccos(vector1 •
vector2) + arccos(vector2 • vector0);

if( abs( angle - 6.28318531 ) < epsilon )
  return true;
else
  return false;
}
```

If the collision detection returns false, the player will move to the new position. Otherwise, the player will stay at the original position.

3.3.4 3D Rendering

The game engine uses OpenGL to perform 3D graphics rendering.

CHAPTER FOUR

3D MODELING

4.1 Overview

A 3D model is an object presented by a 3 dimension polygon. It is mostly created by 3D model software, such as 3D Studio Max, Maya, AutoCAD, and Blender. Basically, a 3D model is rendered by a wireframe and it could be colored with a 2D texture image.

Figure 11. Wireframe of a Cube and a Cylinder

A texture is a normal digital image which can be easily created by any graphics software, such as Adobe Photoshop or GIMP. However, by mapping a 2D texture can make the 3D model more realistic. The mapping process is called UV mapping, which is using a 2D image to express a 3D model. In contrast to (X, Y, Z) is the coordinates of a 3D object, (U, V) is the coordinates of the 2D texture image.

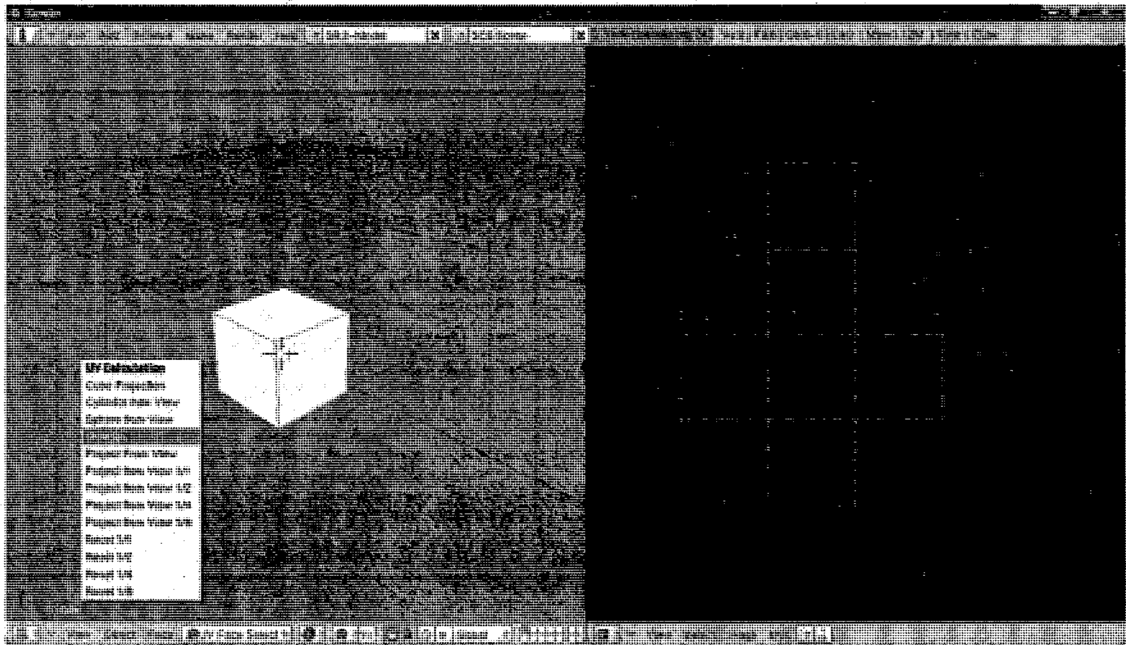

Figure 12. Unwrap 3D Objects with Blender

4.2 Blender

Blender is free open source software for 3D model. It takes less installation size than other 3D modeling software and runs on several different operation systems, such as Microsoft Windows, GNU/Linux, Mac OS X, and FreeBSD.

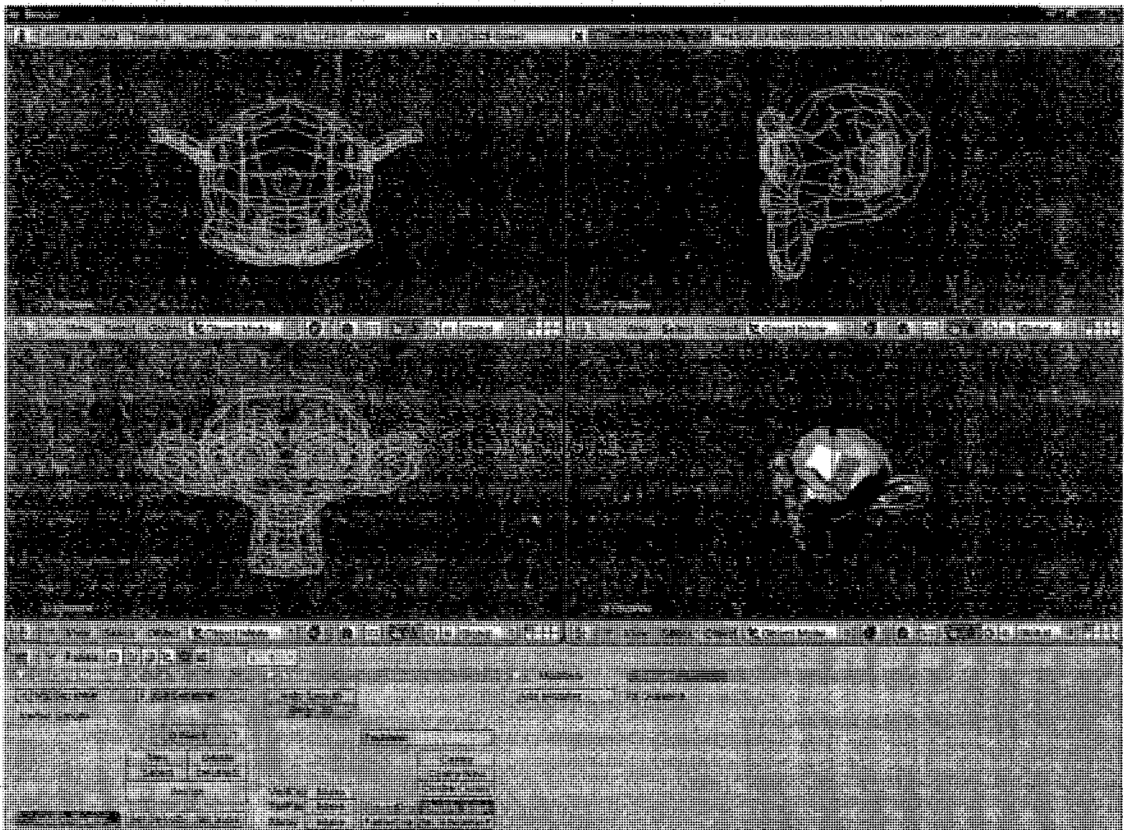

Figure 13. Blender's User Interface

Blender provides many tools for 3D modeling. Therefore, Blender has a complex user interface. Basically, Blender's

user interface has three features: editing modes, hotkey utilization, and workspace management. Editing mode is one of the primary modes to modify a 3D model. Another primary mode is object mode. You can switch between these two modes with the Tab key. Not only switching modes uses the Tab key, most commands in Blender use hotkeys. With workspace management, Blender has more flexible workspace. Figure 3 shows that the Blender' user interface can be split up into four different views: top view, front view, side view, and camera view. Users can easily change the size of each window or merge two windows into a large one.

The newest version of Blender is Blender 2.43, released on February 18, 2007. However, the project is still using last version, Blender 2.42a, to prevent any conflict between two different versions.

4.3 Construction of the Project World Model

4.3.1 Jack Brown Hall

To construct the project world model, the first thing is to get the blueprint of Jack Brown Hall, and then use the image as the background with Blender. Figure 14 shows the top-view of Blender after loading the background image.

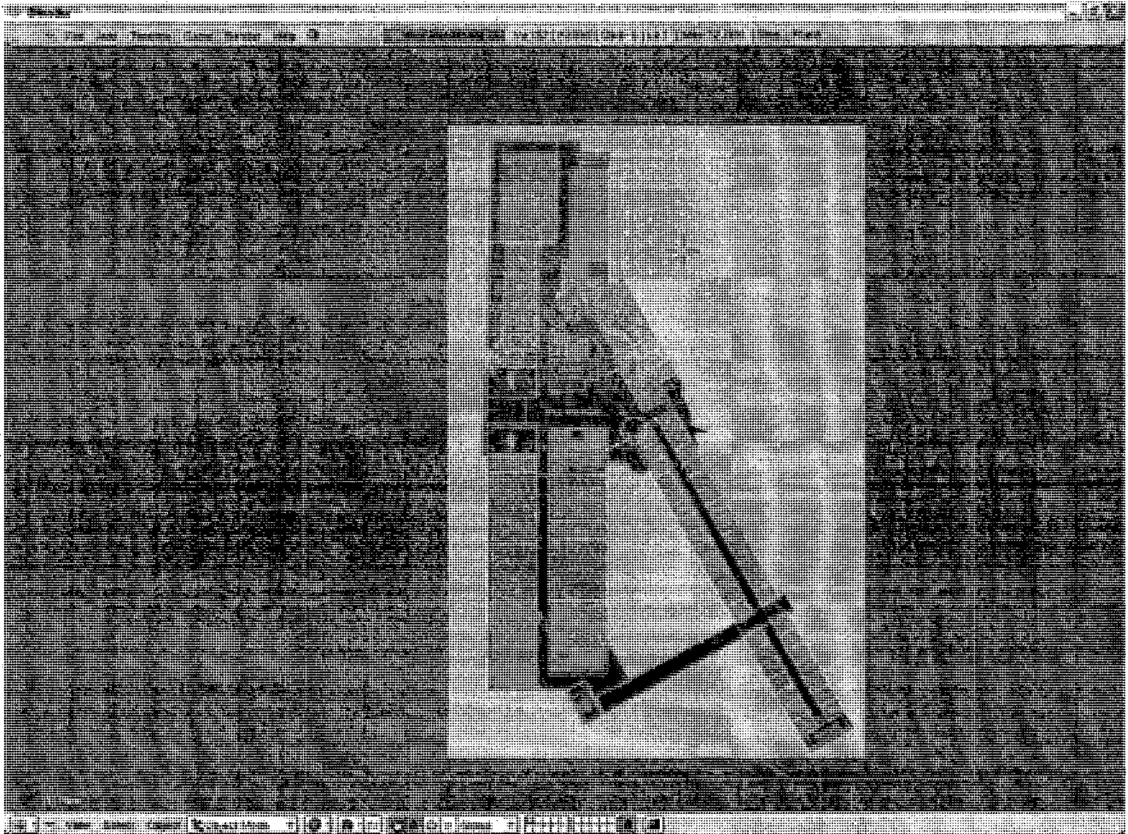

Figure 14. Top-view with a Background Image

After that, we have a rough sketch of Jack Brown Hall. The next thing we need is to make the walls have thickness. In Edit Mode, press BKEY to select the vertices of the wall which needs to be modified. Figure 15 shows that the selected vertices are yellow, and the others are pink.

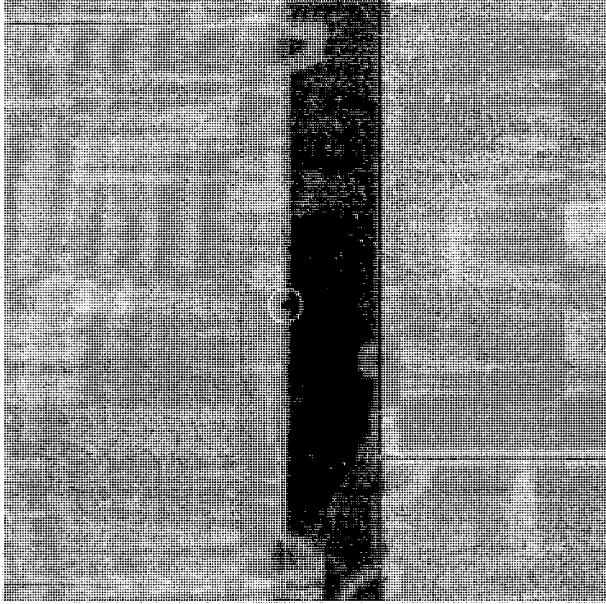

Figure 15. Select Vertices

Then, press SHIFT + DKEY to duplicate the selected vertices, and move these duplicated vertices by the side of the wall. The distance between the selected vertices and the duplicated vertices will be the thickness of the wall.

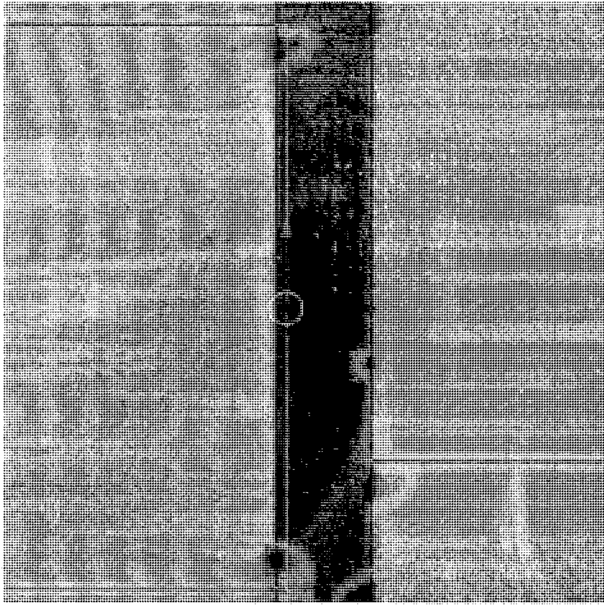

Figure 16. Duplicate Vertices

After modifying each wall, the next step is to make the height of the wall. In Edit Mode, press AKEY to select all vertices, and press NumPad 1 to change the view to front-view. Then, press EKEY to extrude the wall. After that, we have a basic 3D model of Jack Brown Hall.

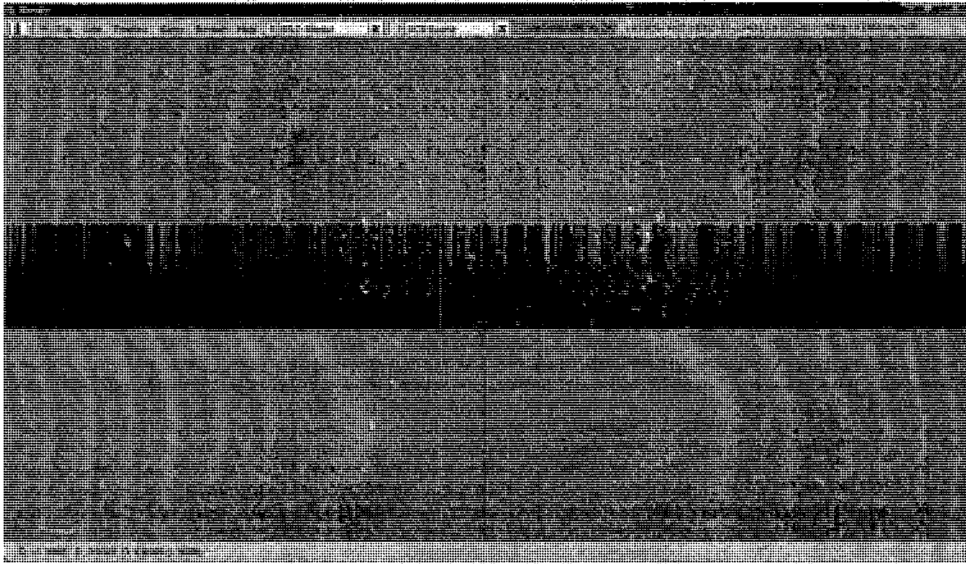

Figure 17. Extrude Object

Figure 18. Main Jack Brown Hall Model

4.3.2 Ceiling and Floor

The 3D models of the ceiling and the floor of Jack Brown Hall are very easy to build. All I need to do is to add a plane in Blender and go to Edit Mode to modify the shape exactly as same as Jack Brown Hall model. The following picture shows the 3D model of the floor.

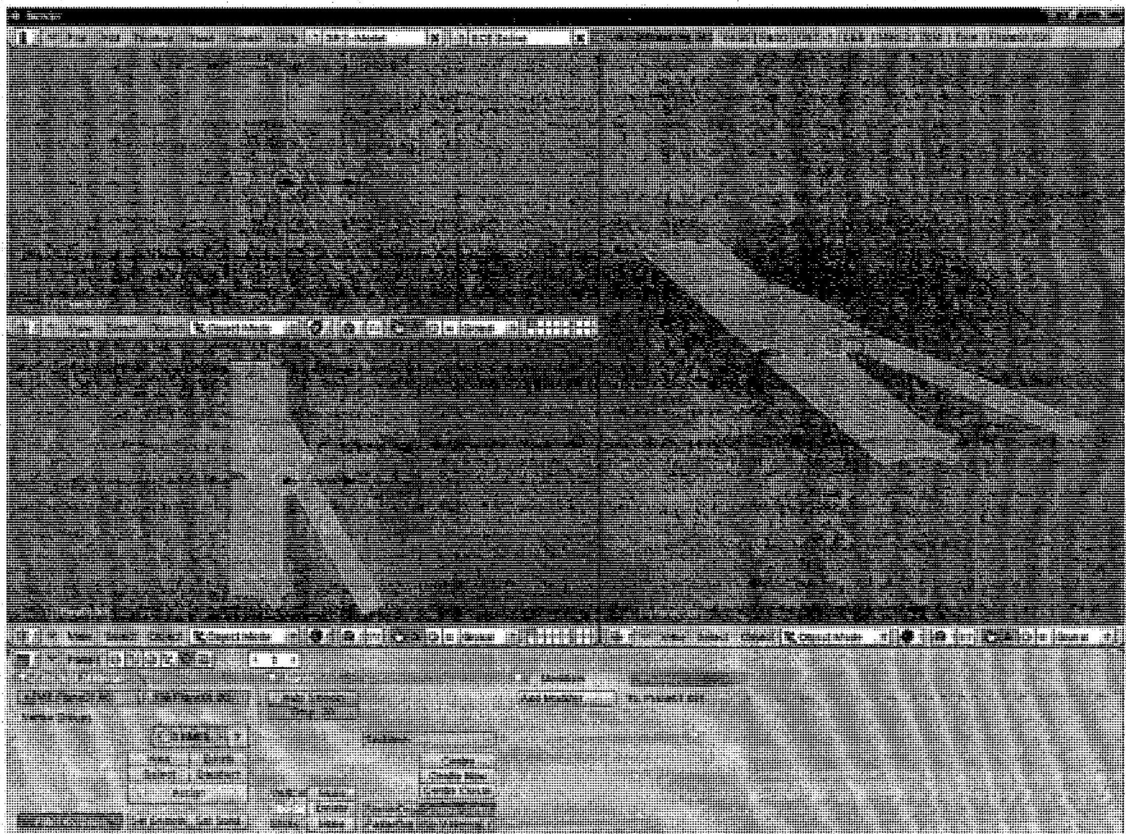

Figure 19. The 3D Model of the Floor

4.3.3 Doors

Blender has a build-in game engine. With this engine, we can make a simple animation to open and close a door. First, press space to add a cube and build a door at frame 1. Figure 20 shows that the door is closed at frame 1.

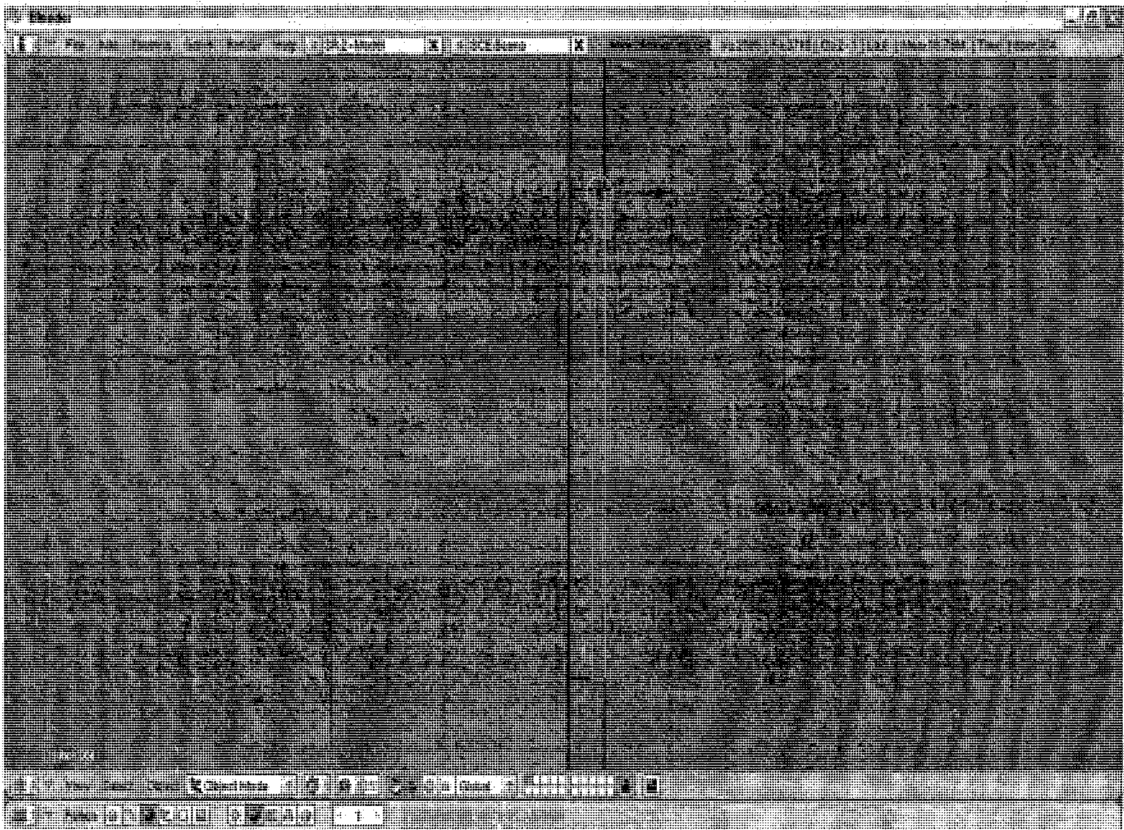

Figure 20. Closed Door

Next, change the frame to 51 and press IKEY to choose "Insert Keyframe" as "Rot" (rotation). Then, press RKEY to

rotate the door to the opened position. The following picture shows the opened door at frame 51.

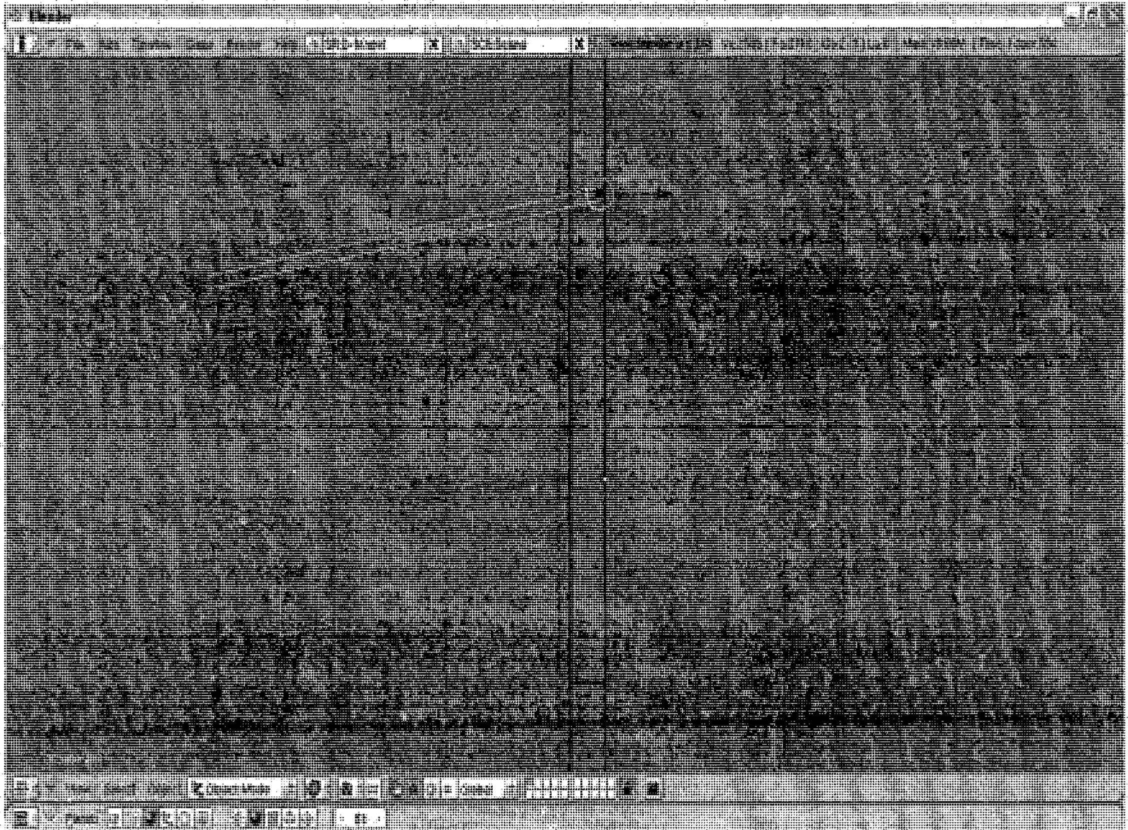

Figure 21. Opened Door

After building the door model, we need to use the build-in engine in Blender to perform the animation. First, press F4 to open the logic panel and add two sensors. The first sensor I set is the space bar, and the second sensor is the distance between the player and the door, which is 0.5. Then, add an

actuator to play the animation which is started at frame 1 and ended at frame 51. Figure 22 shows the whole setting of the door animation.

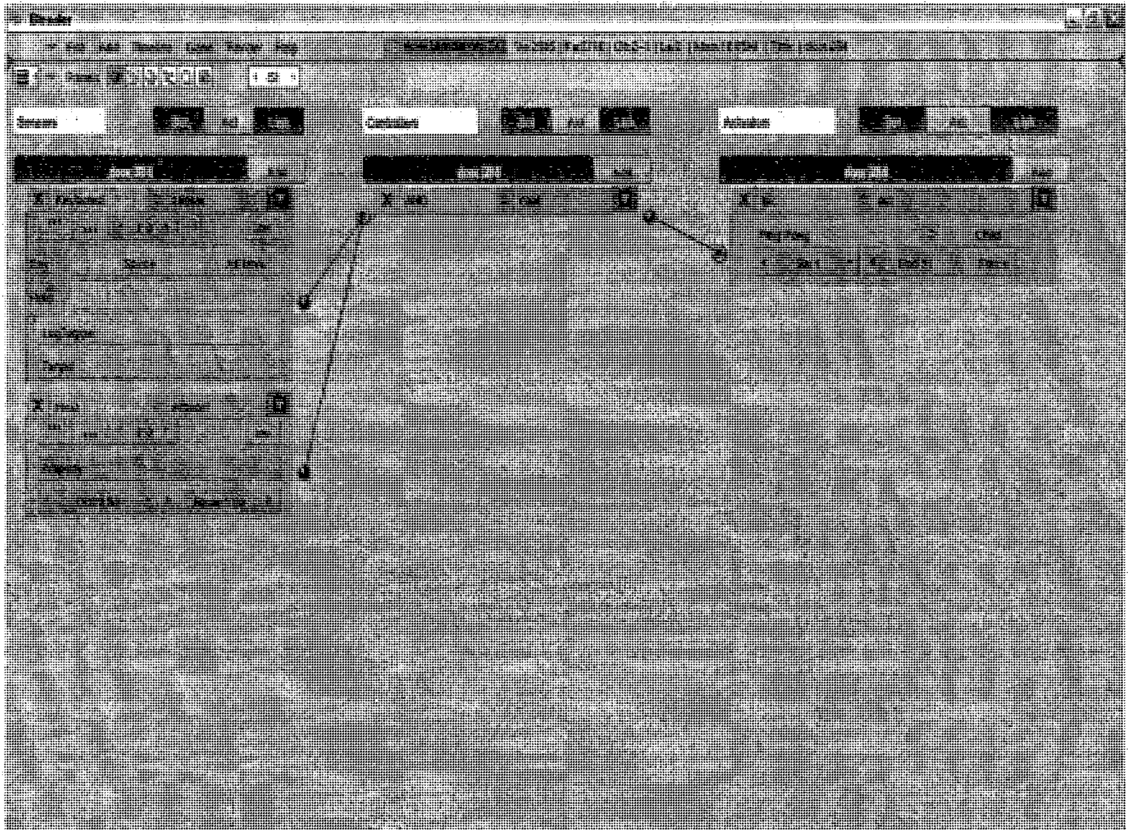

Figure 22. Door Animation Setting

With this logic setting, the door will be open when the player closes to the door and press the space bar.

4.3.4 Sky

There are many ways to build a skybox in Blender. This project is using a half sphere to build a skybox. First, press Numpad 7 to switch to top view. Then, press space bar to add 'Mesh > IcoSphere'.

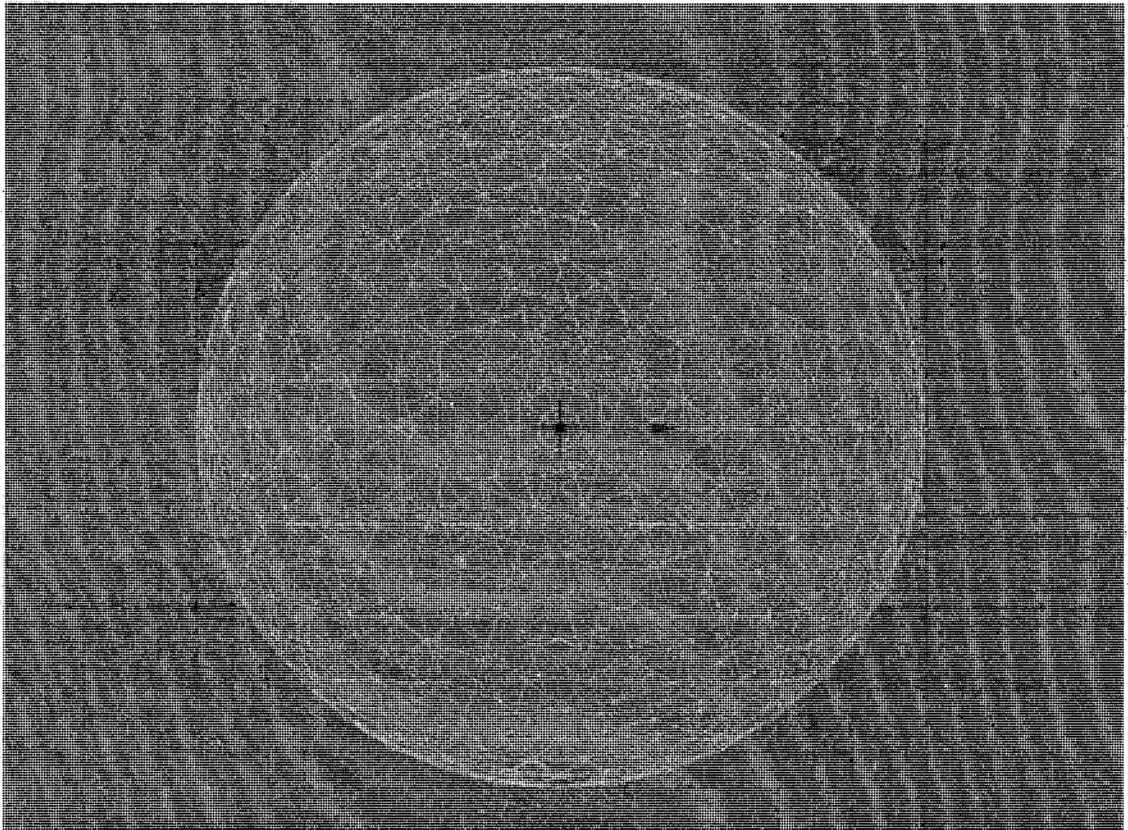

Figure 23. Icosphere

After that, we need to cut the sphere in half. First, press Numpad 1 to switch to front view. Next, press BKEY and select

all the bottom vertices. Then, press XKEY to erase these selected vertices. Now, we have a half sphere.

However, the face of this half sphere is facing outward, which is the wrong way. To fix this, we need to flip over this half sphere to make it face inward. First, press AKEY to select all faces, and then press WKEY to select 'Flip Normals'. Now, we fix the problem and have a basic skybox model. Chapter 4.3.6 will show how to add a texture to the skybox.

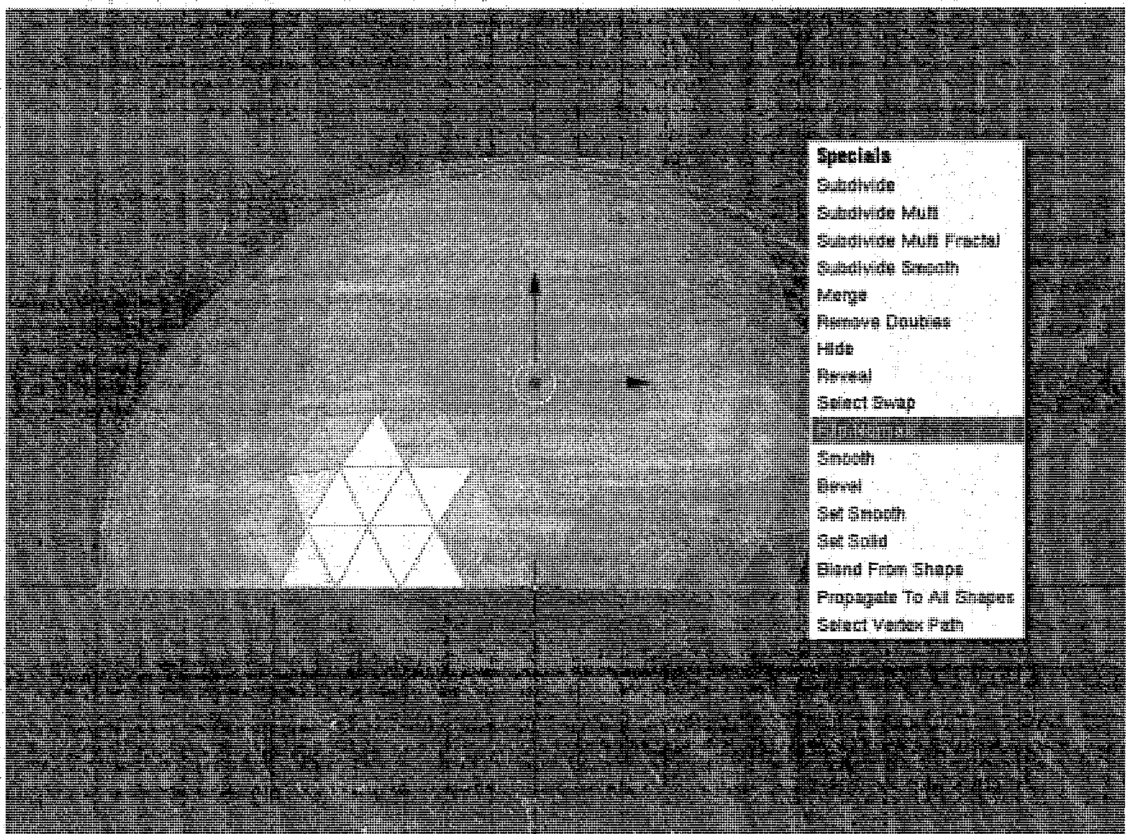

Figure 24. Flip Normals

4.3.5 Ground

After building the sky model, we need to build the ground model as well, or other models will look like flying in the sky.

It is simple to make a ground model. First, press Numpad 7 to switch to top view and press space bar to add a plane. Next, press WKEY to select 'Subdivide' and cut the plane into several pieces. Now, we have a flat ground.

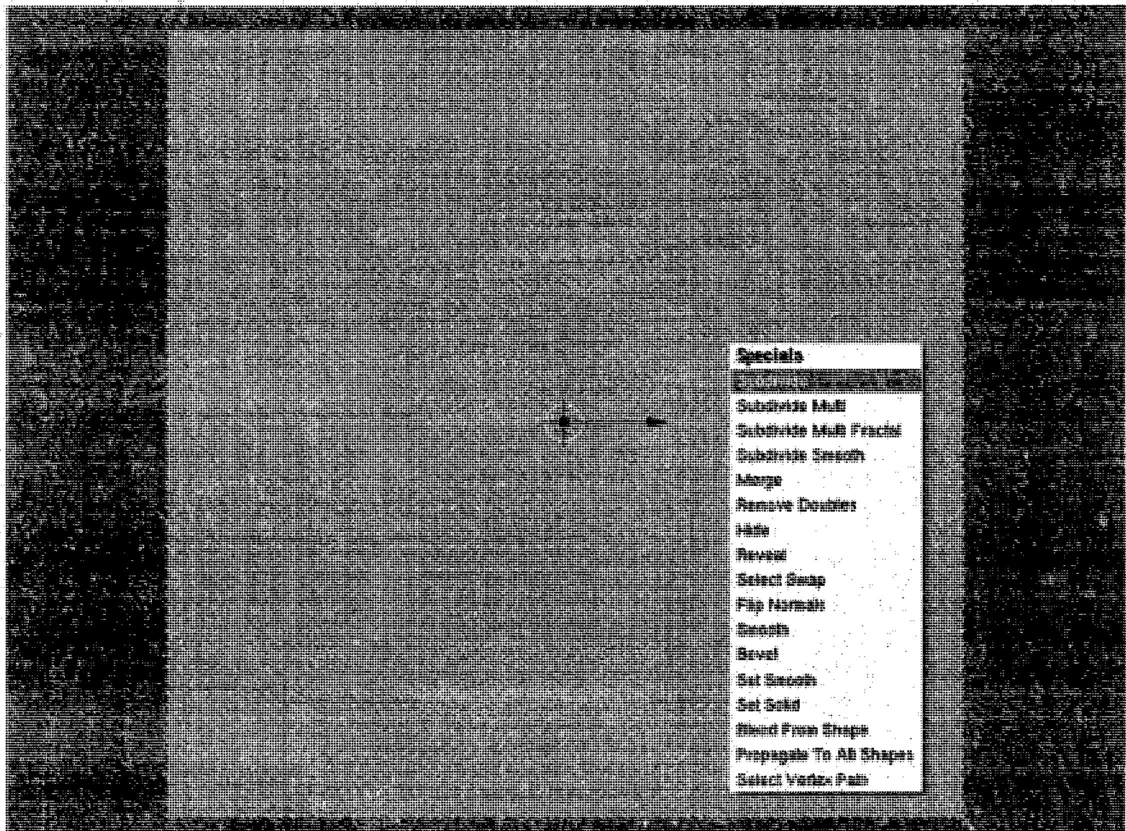

Figure 25. Subdivide a Plane

To make the ground more realists, we can make mountains in this plane. First, right click to select a vertex as the top of a mountain and then press Numpad 1 to switch to front view. Next, press OKEY to enable 'Proportional Edit Falloff' and select one falloff type. Then, left click on the z-axis of the vertex to move the vertex, and use the mouse wheel to change the falloff range.

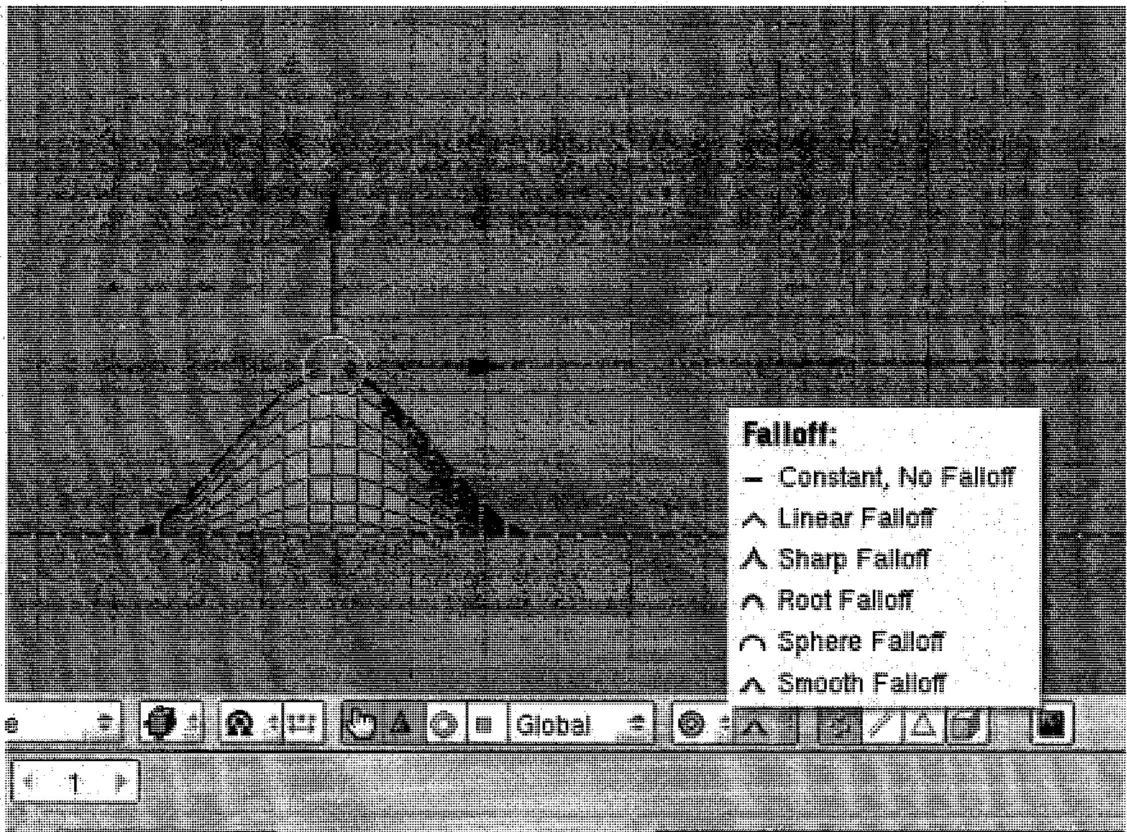

Figure 26. Proportional Edit Falloff

By repeating these steps, we can build many mountains in the ground model. Figure 27 shows the ground model I made in this project.

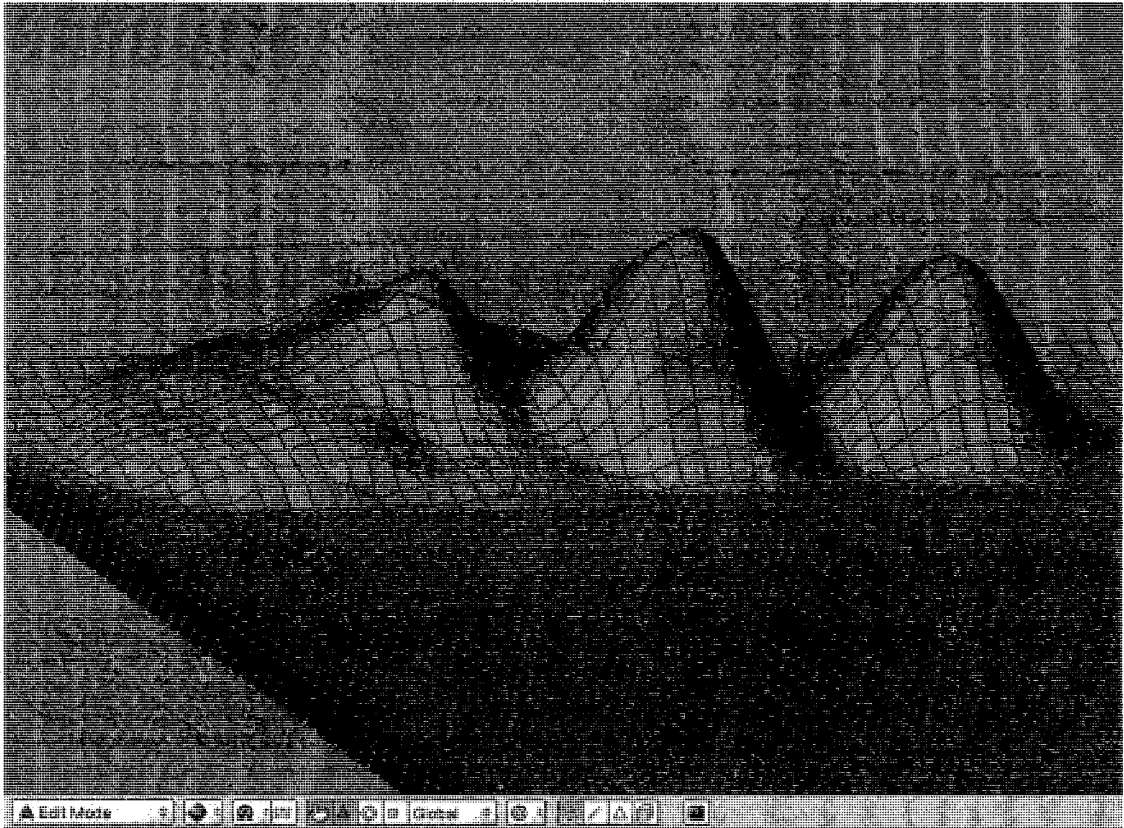

Figure 27. Ground Model

4.3.6 Textures

After finishing these 3D models, the next step is to add textures. Since UV mapping is mapping a two-dimensional texture over a three-dimensional object, the first thing we

need to do is to unwrap a 3D object into a flat 2D object. Here is an example of UV mapping.

This example is how I apply a texture on the sky model. First, select the sky model and switch to the 'UV Face Select' mode.

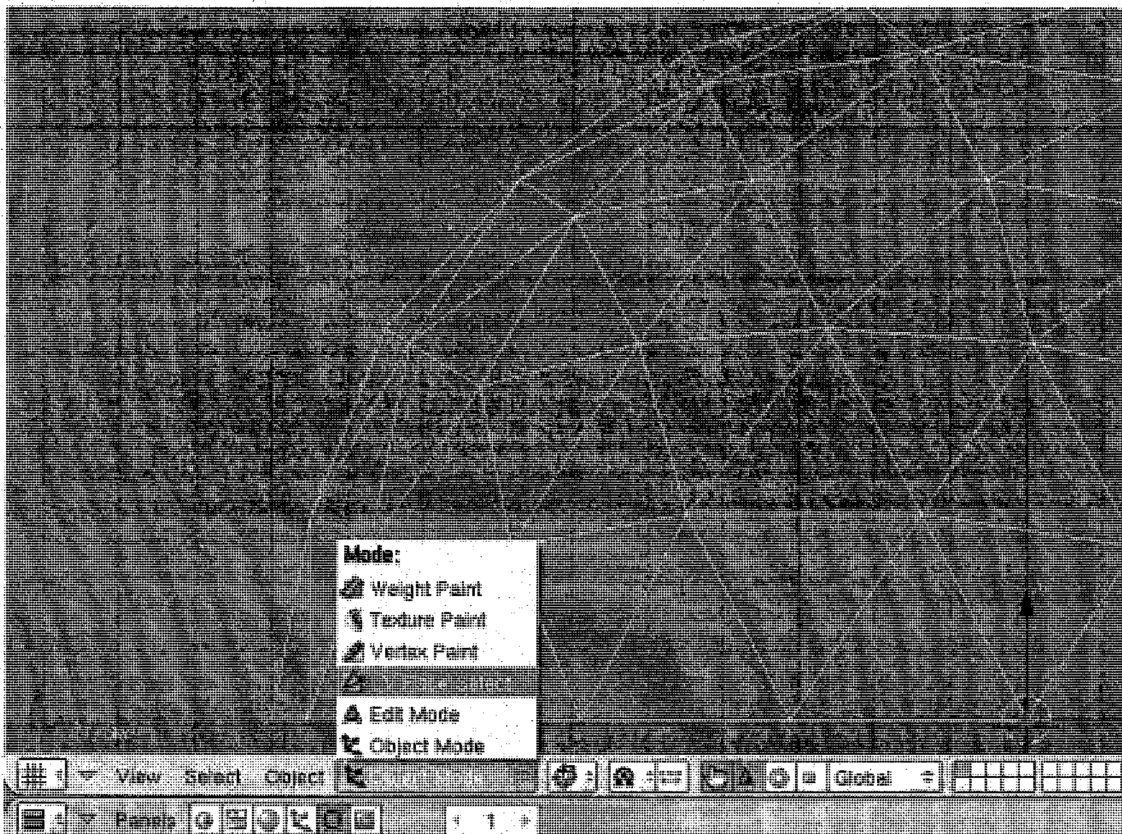

Figure 28. Face Select Mode

Then, press AKEY to select all faces, and press UKEY to select 'Cylinder from View'. Usually, from now on, I would like

to split the window into two different views. One is 3D view in 'UV Face Select' mode, and another one is 'UV/Image Editor'. The following picture is the screenshot of the UV editor in Blender.

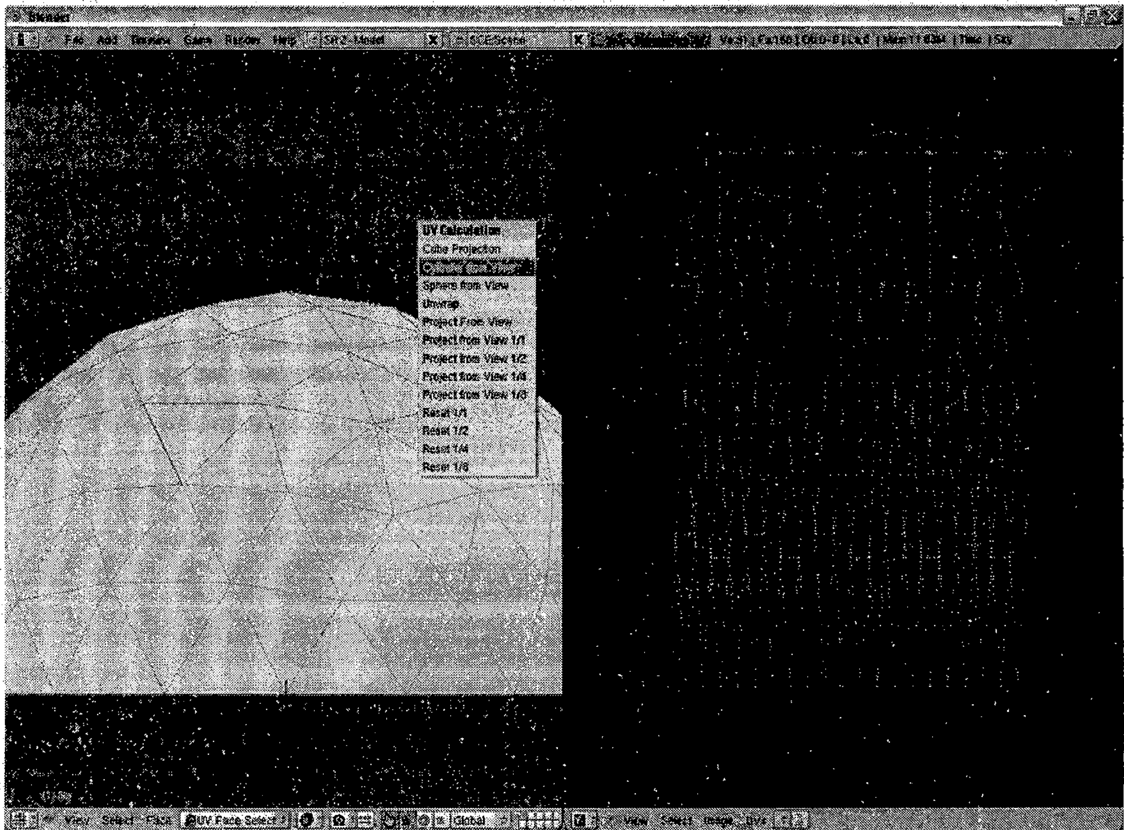

Figure 29. Image Editor

Now, the 3D model is already unwrapped into a 2D object. Next step is to apply an image on it. First, in the 'UV/Image Editor' window, select 'Image' and open the image file. However,

the image file and the 2D object are not perfectly matched, so we need to modify the size and scare of the 2D object in order to get the better result. Figure 30 shows the result of the sky model after UV mapping.

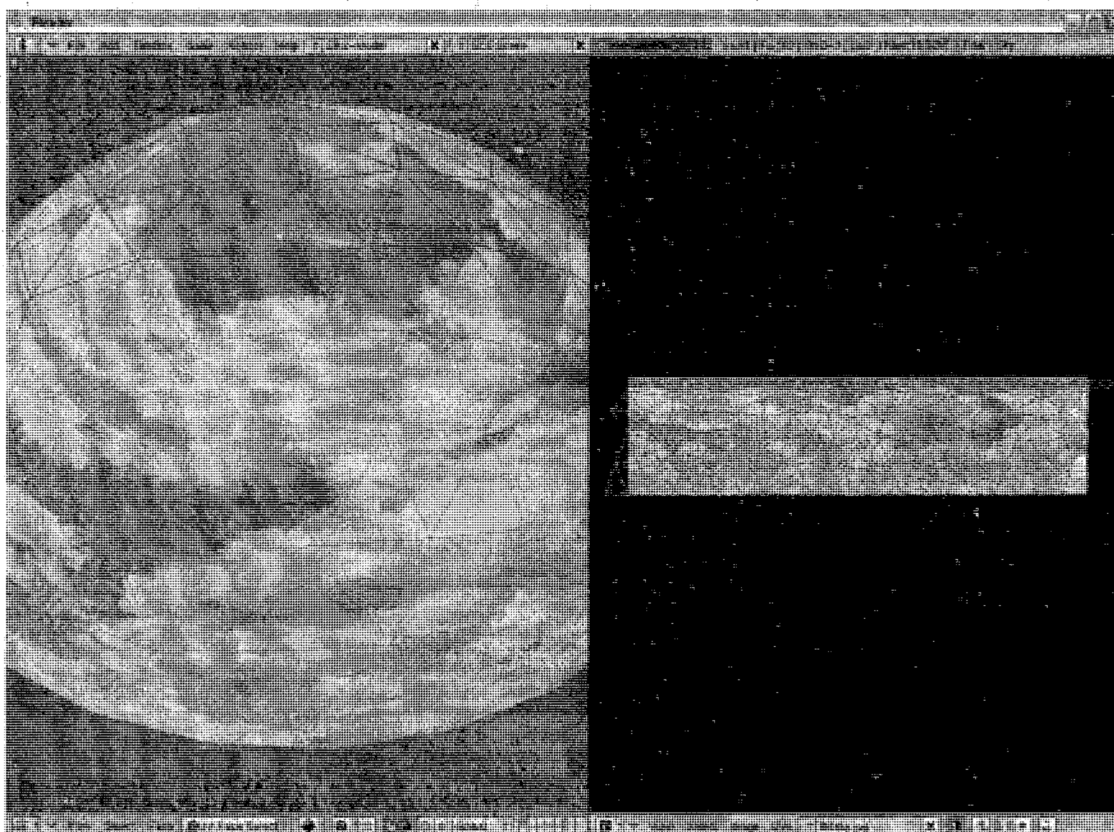

Figure 30. Sky Model

4.4 Construction of the Character Model

4.4.1 Character Model

The character model in this project is also built by Blender. The first step is building a head model. I started with adding a plane in the front view. Then, subdivide the plane into twelve pieces. The following picture is the front view of the plane.

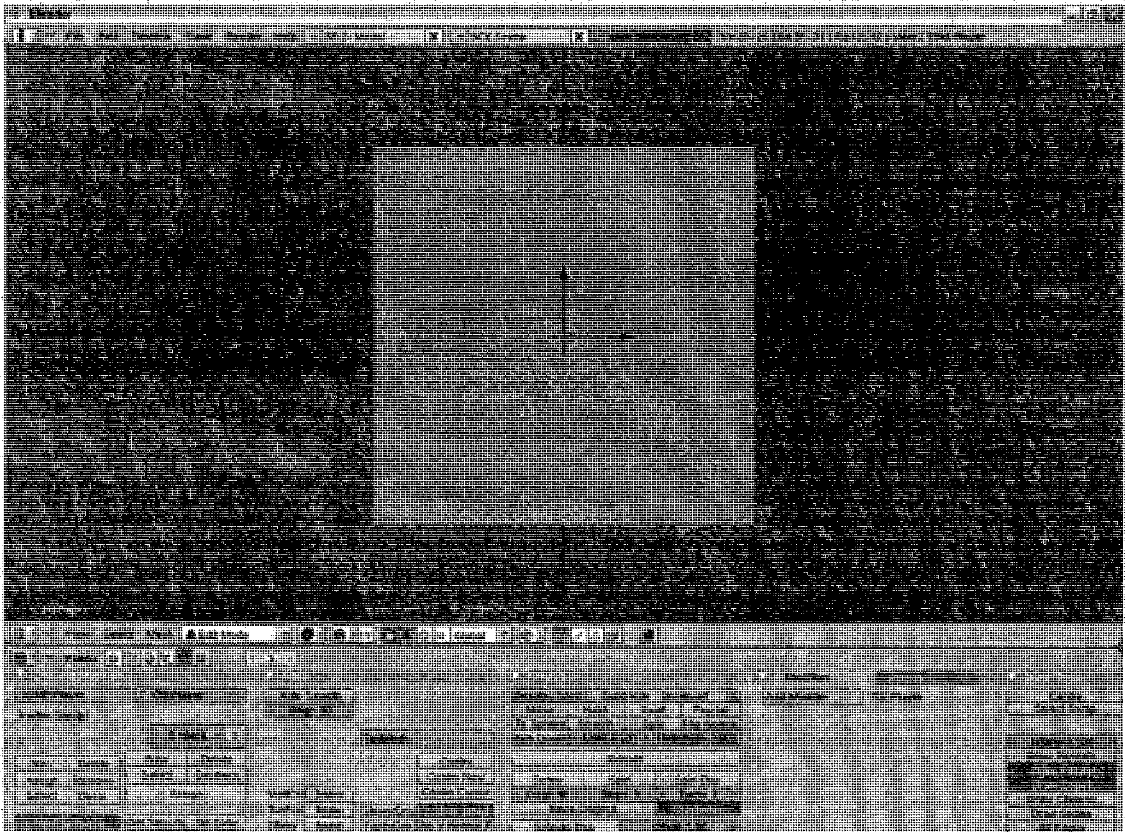

Figure 31. Subdivided Plane

This plane will be used for the face model. I will use the top three squares as a forehead, next three squares as eyes, next three squares as a nose and cheeks, and the last three squares as a chin. After modifying the position of these vertices, the plan will look like the picture below.

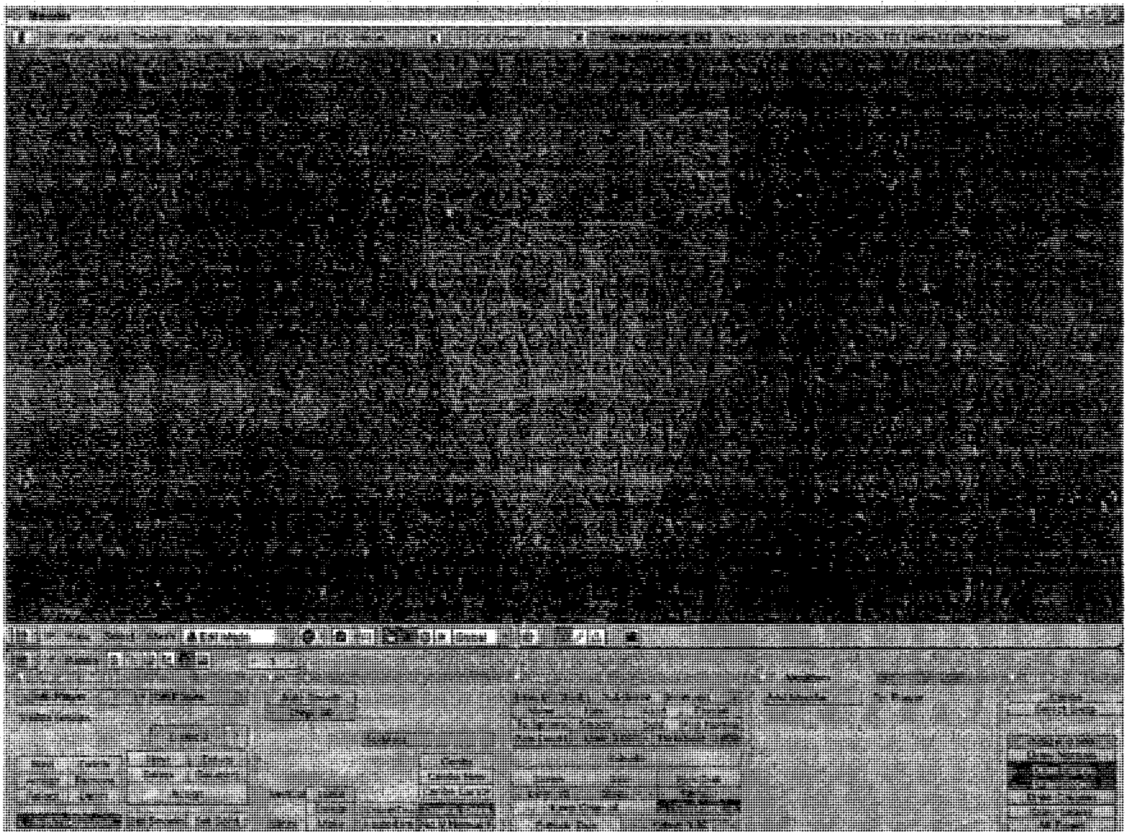

Figure 32. Face Model

After that, switch to side view and move vertices along Y-Axis to perform a nose and eyes. Figure 33 shows the complete face model.

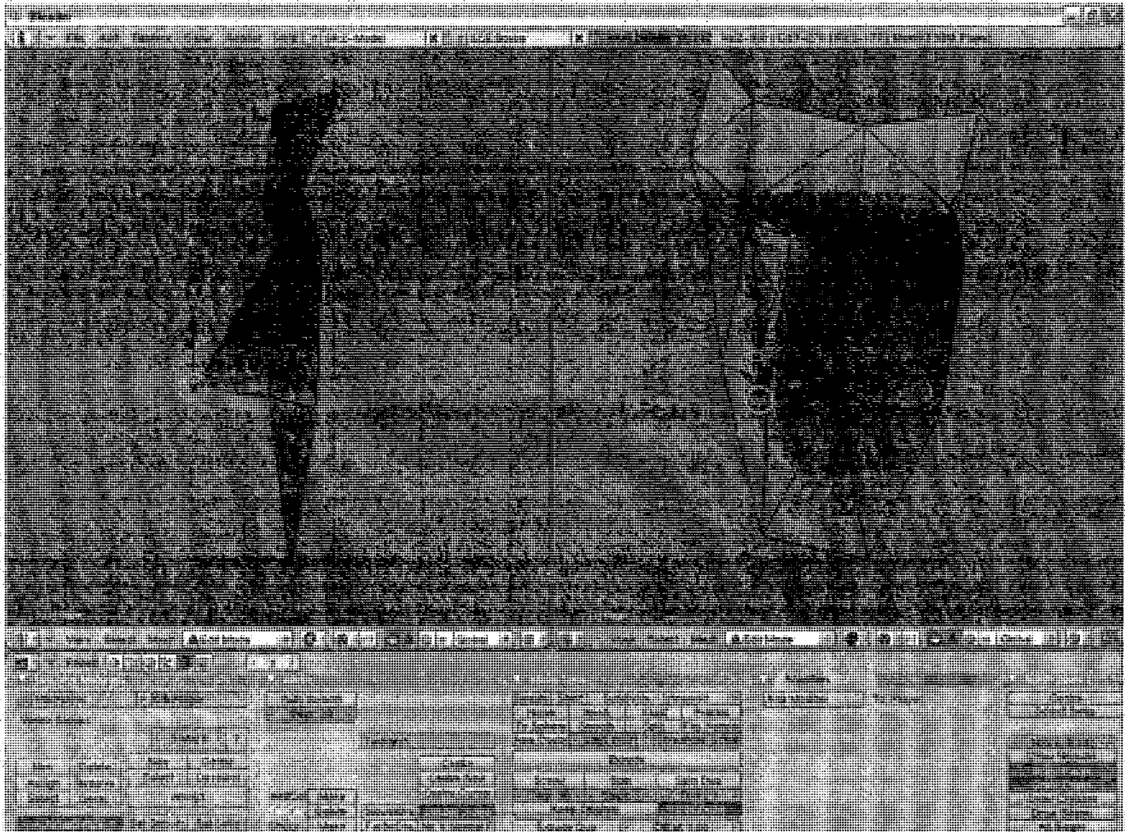

Figure 33. Complete Face Model

By repeating these steps, I finished the whole head model. Next, I need to build a body model. A body model is much easier than the head model. I started with a box model and extruded

each side to be hands and legs. Figure 34 is the complete character model.

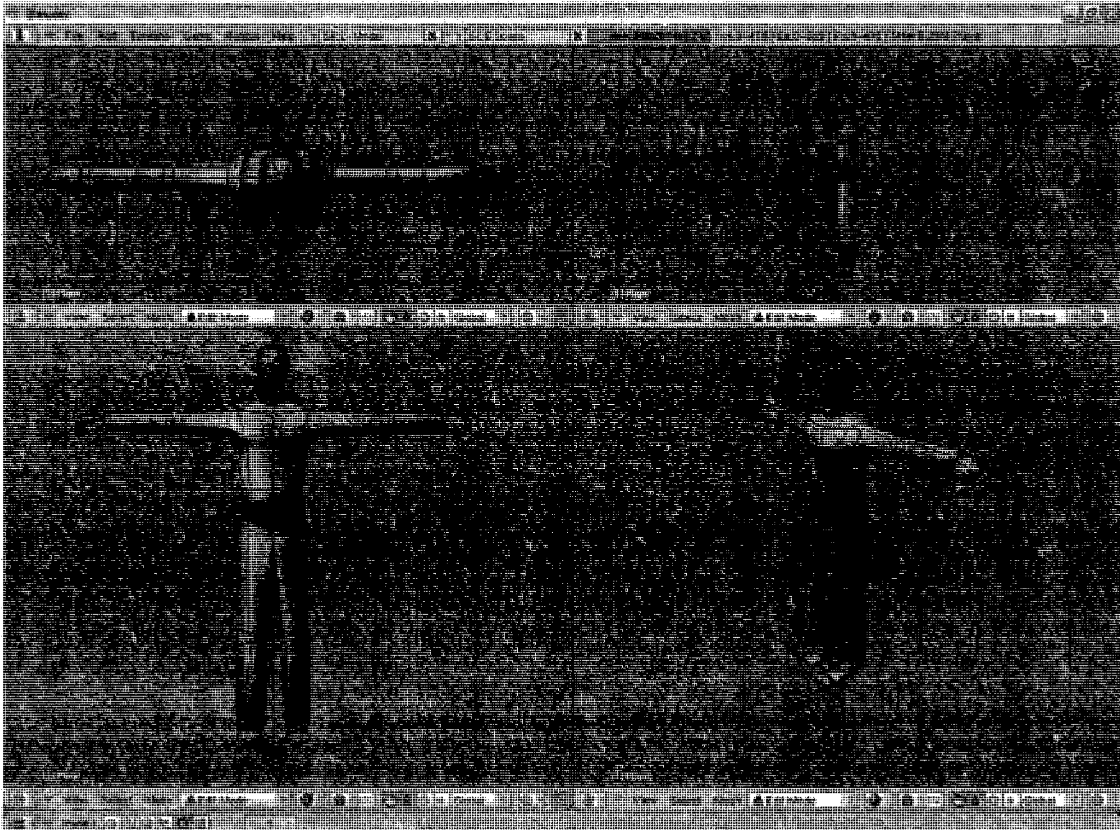

Figure 34. Character Model

Like other 3D models, the character model also needs textures. Therefore, the last step is using UV mapping to apply the image. Figure 35 shows the screenshot of UV mapping.

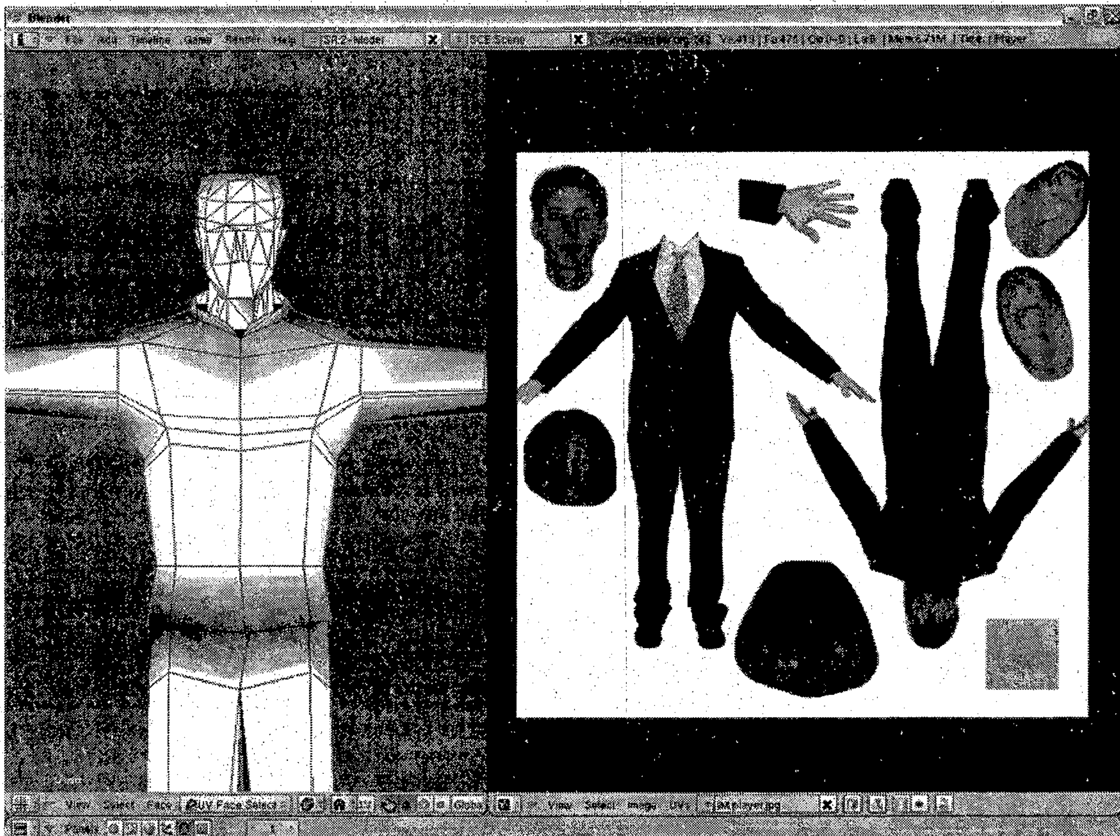

Figure 35. Character Model Textures

4.4.2 Armature

Armatures are like skeletons inside the human body. We can use armatures to pose and deform 3D models. Armatures are usually used in character animation, but they are also very useful in building models, such as a winding pipe.

Like other objects, an armature has a center, a rotation, and a scale factor. It also has Object Mode and Edit Mode. Unlike other objects, an armature has a special mode called Pose Mode.

In Pose Mode, each bone can be moved, scaled, or rotated to perform a particular pose.

After building the character model, we need to add an armature for the character so that we can easily modify the pose of the character. Here is an example of using an armature. First, press space bar to add an armature in Object Mode. Then, modify the position, angle, and scale of the first bone. Next, press space bar to add other bones and also name every bone. Assigning names to the bones is very important because it will make everything easier when editing actions. Figure 36 shows the names of bones used in this armature.

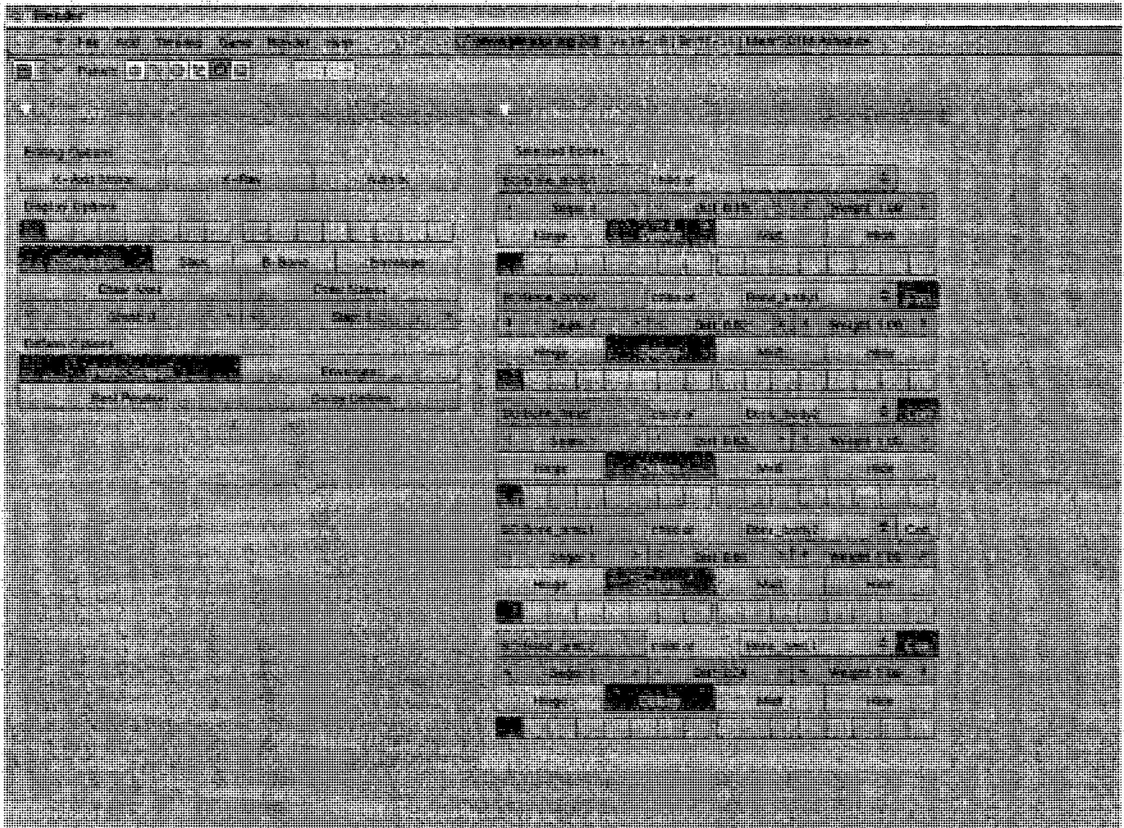

Figure 36. Assigning Names to the Bones

After assigning names to each bone, adjust the position and the scale of each bone to fit in the character model. Figure 37 is the complete armature.

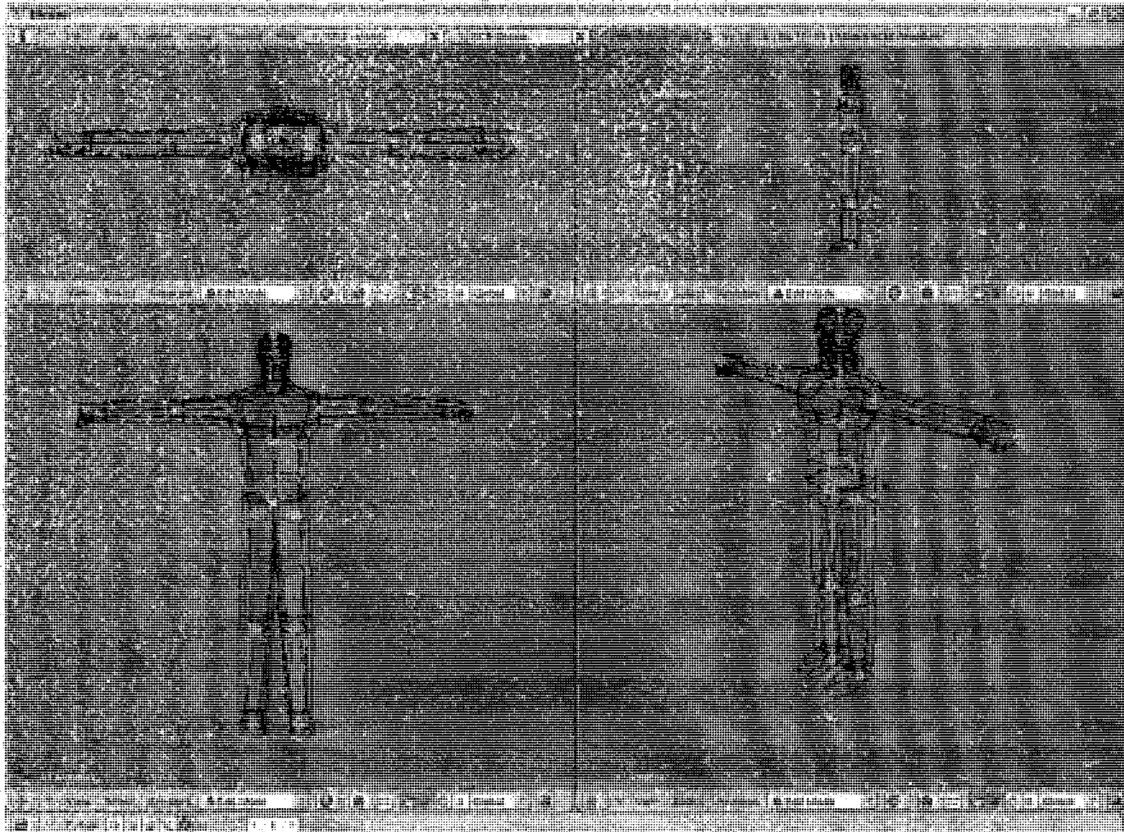

Figure 37. The Complete Armature

After finishing the armature, the next step is to connect the character model and the armature. First, select both the character model and the armature. Then, press Ctrl + PKEY and select 'Make Parent to Armature'. Now, we have to assign vertices of the character model to the particular bone. For example, the vertices of the right arm should assign to the bone of right arm. Figure 38 shows that the vertices of the front right arm are assigned to the bone called 'Bone_armR2'.

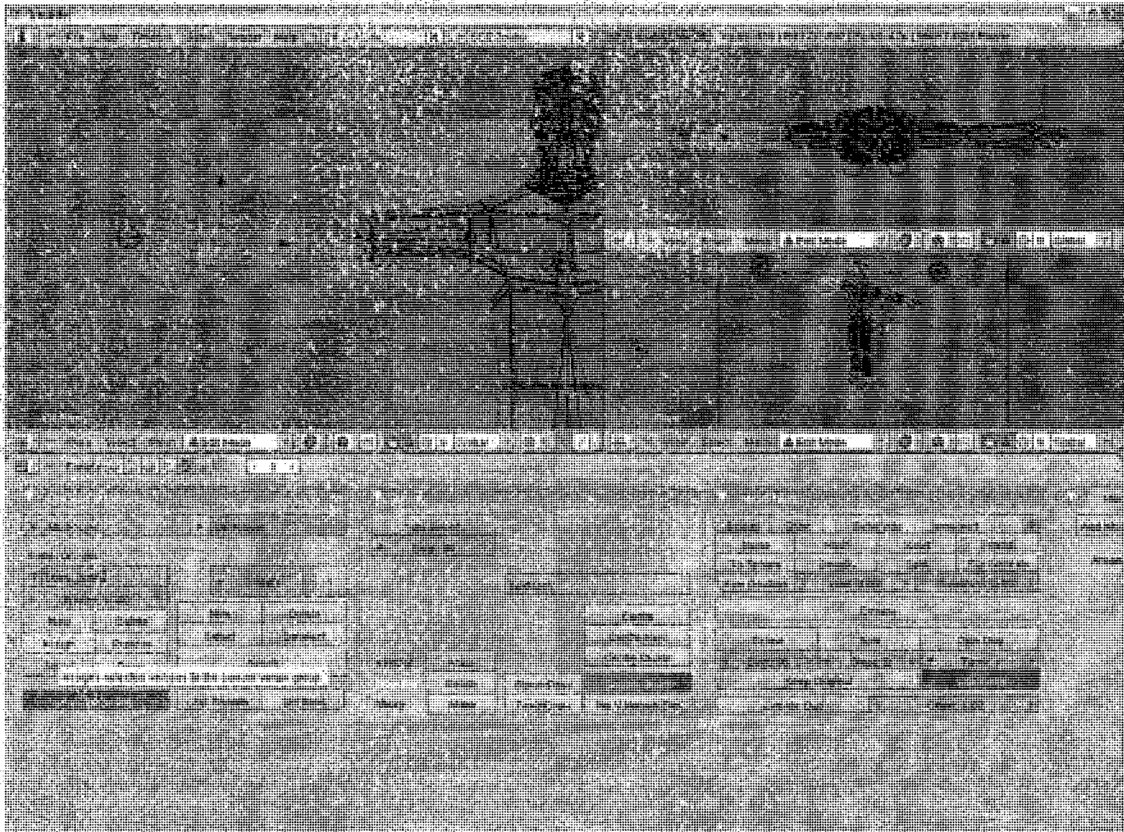

Figure 38. Assigning Vertices to a Bone

There are 11 bones in this armature. After assigning vertices to a particular bone, the character model is able to change any pose we want. First, select the armature and switch to Pose Mode. By moving, rotating or scaling these bones, we can make the character model to do any action. Figure 39 shows the walking pose of the character model.

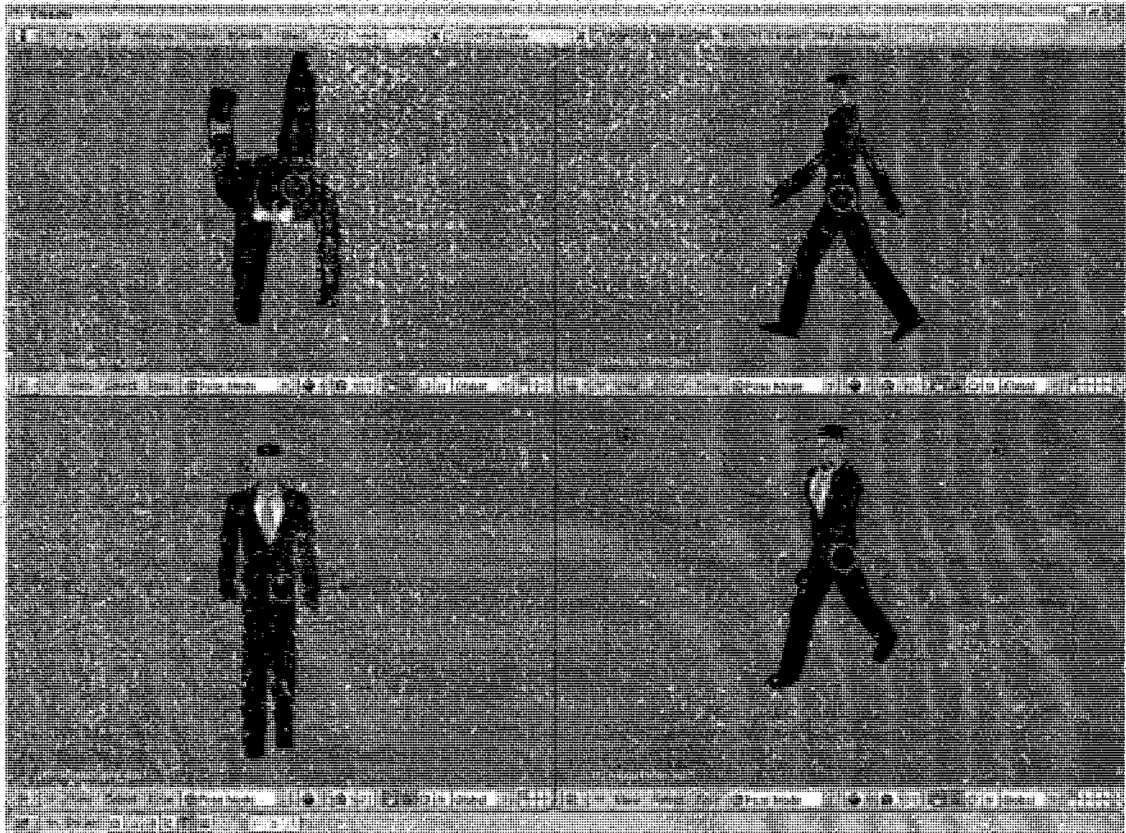

Figure 39. Walking Pose

CHAPTER FIVE

CONCLUSION AND FUTURE DIRECTIONS

5.1 Conclusion

Game development covers several different fields of studies. Basically, there are at least three roles in the developing team: programmers, game designers, and 3D modelers. In this project, developers need to work together and use different tools to develop a game. Some of the tools are available for free; other tools were built specifically for this project. The already available tools include Blender and GIMP. The tools that were built specifically for this project include the WorldStudio level editor, the 3D model conversion tool, and the terrain generator.

The game production process in this project has three steps. First, 3D modelers create art assets. Second, 3D modelers use the tools developed by programmers to preprocess these art assets. Finally, game designers use the lever editor, WorldStudio, to perform the game.

5.2 Future Directions

This project is a very good start for anyone who interests in game development. So far this project only allows users walking around the hallways and sending each other text

messages. The project can be extended by adding more art assets, such as tables, chairs, computers, blackboards and windows. The quality of player models could also be improved by adding more texture and detail design. However, it will need a better 3D graphic card to run the game. Also, adding some plots in the game mode is a good idea. A good story is always an important element in a successful game.

REFERENCES

[1] 3DStudio File Format, http://www.the-labs.com/Blender/3DS-details.html

[2] Alexander Nareyek, "Artificial Intelligence in Computer Games—State of the Art and Future Directions", ACM Queue, vol. 10, pp. 58-65, February 2004.

[3] Alexander Nareyek, "Game AI is Dead. Long Live Game AI!", IEEE Intelligent Systems, vol. 22, no. 1, pp. 9-11, Jan/Feb 2007.

[4] Blender 3D: Noob to Pro, http://en.wikibooks.org/wiki/Blender_3D/Noob_to_Pro

[5] Blender Artists Forums, http://blenderartists.org/

[6] Blender Home Page, http://www.blender.org/

[7] Blender Nation, http://www.blendernation.com/

[8] Daniel Sanchez-Crespo, "Core Techniques and Algorithms in Game Programming", New Riders Games, first edition, September 2003.

[9] GIMP - the GNU Image Manipulation Program, http://www.gimp.org/

[10] OpenGL, http://www.opengl.org/

[11] Wikipedia, Game Engine, http://en.wikipedia.org/wiki/Game_engine

[12] Wikipedia, Plane (mathematics), http://en.wikipedia.org/wiki/Plane_(mathematics)